Application-Oriented Optimizations for Embedded Systems: Architecture, Tradeoffs, and Case Studies

by

Josh Cory Pfrimmer
B.Eng., University of Victoria, 2002

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF APPLIED SCIENCE

in the Department of Electrical and Computer Engineering

© Josh C. Pfrimmer, 2005
University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part, by photocopy or other means, without the permission of the author.

Application-Oriented Optimizations for Embedded Systems: Architecture, Tradeoffs, and Case Studies

by

Josh Cory Pfrimmer
B.Eng., University of Victoria, 2002

Supervisory Committee

Dr. K.F. Li, (Department of Electrical & Computer Engineering)
Supervisor

Dr. M. W. El-Kharashi, (Department of Electrical & Computer Engineering)
Departmental Member

Dr. D. Rakhmatov, (Department of Electrical & Computer Engineering)
Departmental Member

Dr. H. Müller, (Department of Computer Science)
Outside Member

Dr. J. Muzio, (Department of Computer Science)
External Examiner

Supervisory Committee

Supervisor

Departmental Member

Departmental Member

Outside Member

External Examiner

ABSTRACT

Power efficiency and processor performance demands on modern embedded systems are steadily increasing, and manufacturing improvements are allowing for greater customization of Systems-on-Chip. Meanwhile, the diversification of embedded systems into a growing number of applications, and the continual growth and increasing adoption rate of existing applications drive a trend towards platform based design. Custom and semi-custom platforms allow optimization of hardware to meet the specific needs of individual applications; those specific needs must be determined on a case-by-case basis, but are closely correlated to families of embedded workloads. This thesis presents the embedded system designer with a discussion on available hardware customizations and architectural tradeoffs, categorized in terms of workload and application. Case studies illustrate a process by which the designer can explore such optimizations for their specific application. An in-depth examination is provided for one such tradeoff, on-chip memory configuration, including an empirical analysis of the measurable benefits.

Table of Contents

List of Tables

Table of Figures

Acknowledgement

Foremost, I would like to thank my supervisor, Dr. Kin Fun Li, for his patience, attention, and guidance. I hadn't even considered graduate studies before his invitation, and I am entirely thankful for the inspiration and opportunity.

I am grateful to Dr. Daler Rakhmatov for contributing to and helping to shape the material on scratchpad packing in preparation for a publication submission.

I would also like to thank Mr. Allan Crawford for his generosity and character, and Integrys Limited for their support and the experience it has provided me. I have also received kindness at the hands of Amirix Systems Incorporated, for which I am grateful.

The Natural Sciences and Engineering Research Council of Canada and CMC Microsystems are both supporters of my research, and deserve credit for their generosity and vision. Long may they both continue to support education and research in Canada.

Finally, the I must acknowledge the University that has been my home for eight years of self indulgence and self improvement, and the uncountable staff, teachers, and friends who have made it possible for me.

Dedication

To my parents, whose support and sacrifices have made it all possible. I love you more than you know.

1. Introduction

Colloquially, the term "computer" is used to refer to a desktop or laptop computing device, complete with a keyboard, mouse, monitor, disk drive, and usually network access. Unless specifically modified with further description or with a direct adjective such as "embedded," "integrated," or "micro," (or "mainframe" or "super") this is the first concept that comes to mind when people think of computing. It is surprising to most people, then, that a small percentage of the computers in the world are desktop or laptop, Microsoft-and-Intel (or AMD, or Apple, or Sun) personal computers. In fact, the overwhelming majority of computers in the world today are embedded computers, and the ratio continues to grow in their favor [1].

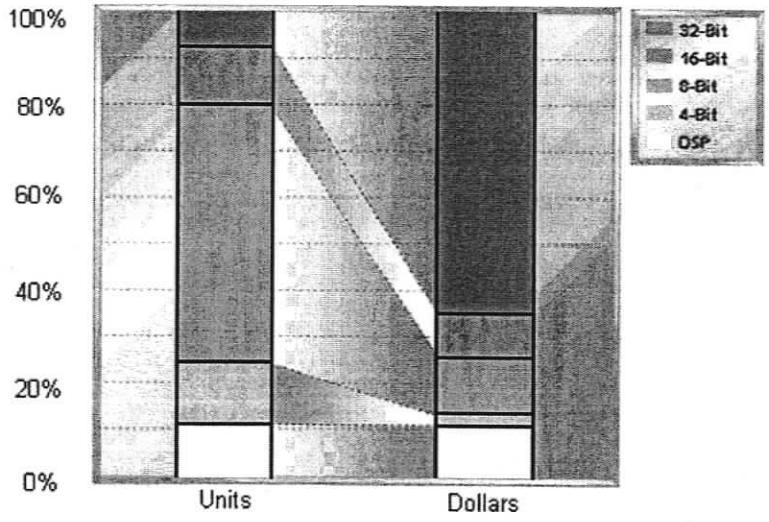

Figure 1.1: Processors and Sales by Type in 2002
Image Source [1]

Figure 1.1 shows the breakdown of processor types by unit count and sales dollars for 2002, according to the World Semiconductor Trade Statistics organization. Despite popular preconception, the class of processors used in the majority of desktop computers, 32-bit processors, accounted for only 9% of the total volume of processors sold in 2002. Even that figure is misleading, though, because there are other uses for 32-bit processors.

In fact, only 2% of the 32-bit processors sold in 2002 were used in personal computers; in all, less than 0.2% of all processors sold in 2002 were used in PCs [1]. Of the remaining processors, some small fraction went into larger computing systems, such as supercomputers, grid processors, mainframe computing, but the overwhelming majority were used in embedded systems.

1.1 What Is Embedded and Why Is It So Popular?

The volume of literature, press, and discussion on the topic has created some confusion as to what exactly constitutes an embedded computer. Before exploring that definition, we will first clarify our use of a couple of terms. Strictly speaking, the word computer may be interpreted to mean only the Central Processing Unit (CPU) of a computer system, which is comprised of many concordant components, including memory and input/output (I/O) devices, amongst others. However, more popularly and perhaps more usefully, computer is used to denote the system as a whole. We adopt this use, and indicate the processing component with the term CPU or processor. Similarly, an embedded computer system is a computer with properties that make it embedded, as we will describe shortly. This term is commonly shortened to either embedded system or embedded computer, interchangeably; an embedded processor is the CPU in an embedded system. We now address the characteristics that classify a computer as embedded.

Literature abounds with vagaries such as "An embedded system is any computer that is a component in a larger system and relies on its own microprocessor" [2], which offer little in the way of explanation. That particular definition can be interpreted to mean that nearly every computer in the world is embedded. The term is generally used by the population at large as a catch-all for any system containing a microprocessor that is not a desktop or laptop PC, defining embedded computing by a small list of what it is not: an embedded computer is not a home or office PC, a workstation, or a supercomputer.

Experts have a somewhat more specific perspective, but still do not agree entirely. Manfred Schlett, in a 1998 article summarizing the embedded industry [3], noted a

tendency to define embedded systems as those that use microcontrollers rather than microprocessors. As to the distinction between those two classes of CPUs, he suggested that we should "define microcontrollers as having RAM or ROM instead of caches, accompanied by a lot of peripherals. In contrast, we think of microprocessors as having a memory management unit and lots of cache."[3] (In fact, a major area of contemporary research, and one covered at length in this thesis, involves the co-habitation of direct-access SRAM and cache on the same chip, for embedded systems.) Alternatively, Schlett noted that the difference between CPU classes is often thought of in terms of performance; "8- and 16-bit devices are normally called microcontrollers, anything more powerful is a microprocessor". It is a sign of the endemic ambiguity in the field that the term "performance" is used to describe native bit width, when the two are actually only loosely related. Schlett was quick, even seven years ago, to point to the growing demand of embedded 32-bit processors as fatal to such a simplistic definition.

In fact, in this rapidly evolving discipline, any definition based on architectural specifics is doomed to obsolescence. A more durable approach is to relate the field to its objectives by observing the nature of embedded applications. In 1997, Malik et al. noted that "software in [an] embedded system is part of the system specification and does not change once the system is shipped."[4] More recently, in 2003 (the years are relevant in a field so dynamic), Petrov wrote about "the one aspect that differentiates embedded processors from general-purpose processors, namely, advance knowledge regarding their application context."[5] Two important points are made here:

1. Embedded computing and general-purpose computing are not interchangeable. In fact, it is implied that there are two distinct classes of computing, general-purpose and embedded. That assertion will be addressed shortly.
2. It is the static nature of the application, and therefore of the software executed by the processor, that defines embedded computing. Put another way, embedded computers are not programmable (according to the definition below).

Care must be taken to distinguish between programmable and interactive, since the terms are sometimes confused in the vernacular. For example, people speak of

"programming" a VCR to record their favorite program, but do not actually write code for it or change its function. When one assigns a channel and time for the VCR to record, they are actually interacting with the existing software in the VCR by supplying it with data. Similarly, when storing phone numbers into a cellular phone, entering formulae into a calculator, or typing addresses into a web browser, the user is exploiting the interactivity of the program, but not directly supplying instructions to the processor.

A Personal Digital Assistant (PDA) is very programmable, as one can purchase or download new applications and alter the functionality of the device, just as can be done with a PC. Similarly, a game console like Sony PlayStation 2 or Microsoft Xbox really is being reprogrammed every time you pop a new game in. To the casual user, unaccustomed to actually writing and developing their own computer programs, there is little difference between purchasing a software package for a PC, a PDA or a PlayStation™. In all cases, the device changes its functionality as a result of a new set of instructions.

Downloading new ring tones to a cell phone, on the other hand, does not change its functionality: when a call is detected, the phone rings. Whether that ring sounds like Mozart or Britney Spears depends on the data set in the memory location allocated for the ring tone; the data used by the ring program. That is not programming - no new software is applied. Nowadays, it is possible to download simple applications such as games to EDGE or GPRS enabled 2.5G phones. 3G phones have even more programmability, and as cell phones evolve and continue to take on more of the duties of a PDA, the more they resemble general-purpose computers. However, one should still categorize cell phones as embedded systems, since their main use remains voice communication, and that utility is not often or easily altered (reprogrammed).

For the purposes of this thesis, it is suggested that a computer system is embedded and not general-purpose if the functionality is fixed to the user. However, this cannot be an absolute dichotomy; there are degrees of programmability. Instead, it is suggested that the more an end user can alter the application on a device, the less embedded (and more general-purpose) the device is.

In addition to the fundamental property above, embedded systems tend to exhibit other characteristics that distinguish them from other computing systems. The following is a list of common, but not essential or exclusive, properties of embedded computers.

1. **Reduced Processing Demand** – Compared with workstations, multitasking personal computers, and larger computing applications, embedded systems generally require less processing capability, though they may have intensive specific needs.
2. **Lower System Cost** – Especially in commercial electronics, per-unit cost is often a key factor in a design. The success of embedded systems in the rapidly changing marketplace is more sensitive to manufacturing and development costs than other forms of computing.
3. **Small Form Factor** – Many embedded systems are portable, or integrated into other forms. In both cases, decreased size of the total system is not only desirable, but sometimes a design constraint.
4. **Lower Power** – For both reduced heat dissipation and increased battery life, most embedded designs place a high priority on achieving low power operation.
5. **Real Time** – Many, but certainly not all, embedded systems operate under real-time constraints, generating time-critical output based on sampled or streaming inputs.
6. **Simpler Operating System** – Because of the tendency for embedded systems to run single tasks, or a small number of tasks simultaneously (unlike other forms of computing, notably desktop, wherein a large number of threads run simultaneously) the demands on the operating system are reduced. In many systems, no operating system is needed at all. Where an operating system is required, it usually emphasizes critical real-time task switching over any other traditional operating system duties. Such an operating system is known as a Real Time Operating System (RTOS) [6].
7. **Smaller Software** – The applications run on embedded software tend to be simpler algorithms, and are implemented in optimized code, as well as specialized

hardware. As a result, the software footprint is usually a fraction of what may be found in other computing classes.

It is worth reiterating that the above list is neither comprehensive nor absolute. Numerous exceptions to each item can be found within the realm of embedded computing, and so we recommend that only the property of having an a priori fixed application (and the corollary lack of reprogrammability) be used to define the field.

1.2 Platform Based Design

Now that we have a working definition of embedded computing, we turn our attention to the forms that embedded computers can take. Chapter 2 will describe the actual end products, or devices, which contain an embedded computer. For now, we focus on the computer itself, and a quick survey of all delivery platforms for embedded computer devices. In particular, we introduce platform based design as an effective and cost-efficient method for designing embedded systems [7].

Simple systems, or those with modest computing requirements, can be assembled from discrete (Commercial Off The Shelf, or COTS) components on a circuit board. Many components previously intended for the desktop computing market have made their way onto the embedded scene, so a great variety of processors with a wide range of features and specialties are available. In fact, many embedded systems are designed in this way, leveraging the popularity of current or previous state-of-the-art software, compilers, and hardware to provide the most computing power for the least money.

On the other hand, many embedded systems do not require that much processing punch. Figure 1.1 shows us that the majority of computers in the world have native word sizes smaller than 32 bits. On the low end of the spectrum, a system with simpler processing requirements may get by with just a 4- or 8-bit processor and some memory. The extremely low cost of these components (e.g., $0.55 for an 8-bit PIC10F200 [9]) makes this the best solution for low-computation applications. Other systems, particularly those manufactured in high numbers, can consist entirely of specific-purpose hardware, and sometimes no processor at all. An example is a common CD player, which

may contain several chips to control the laser and spindle, and to perform Reed-Solomon decoding, then digital to analog conversion, but no true processor.

However, many products and applications are emerging which require significant processing power not available from such low-cost solutions. In addition, power consumption levels that are tolerated in the desktop market present the dual hurdles of heat generation and battery depletion in wireless or handheld devices. To solve these problems, the industry turns to System-on-Chip (SoC) which contain most if not all of the components of an embedded computer; the processor, cache, main memory, system bus, and any number of other coprocessors and contributors to functionality are manufactured on the same die, packaged, and sold as a whole. Power consumption in a SoC is dramatically lower than in an equivalent system built of discrete components, mainly because of the reduced cost of communicating between components. A one millimeter wire at 1.5 volts consumes drastically less power than a four centimeter PCB trace at 5 volts. Also, component count is reduced, which helps to decrease the overall size and manufacturing cost of the system as a whole, provided that the SoC can be obtained cheaply.

However, that very issue is also the drawback of the SoC approach: what does such a system cost? The complexity of embedded systems, particularly those served by SoCs, is increasing rapidly, with transistor counts in the tens and hundreds of millions and growing. The engineering effort, and cost, required to design the logic and components for such a chip is immense, and the manufacturing costs continue to increase even as technologies improve [10], as indicated by Figure 1.2. To deal with this complexity, the industry is turning to platform based design, in which the major components are grouped together as a solution for a particular class of embedded applications. Once developed, a platform can be re-used by tailoring or adding new blocks, saving the engineers the time and effort required to start from scratch. This intellectual property (IP) re-use paradigm amortizes development costs among a larger number of products.

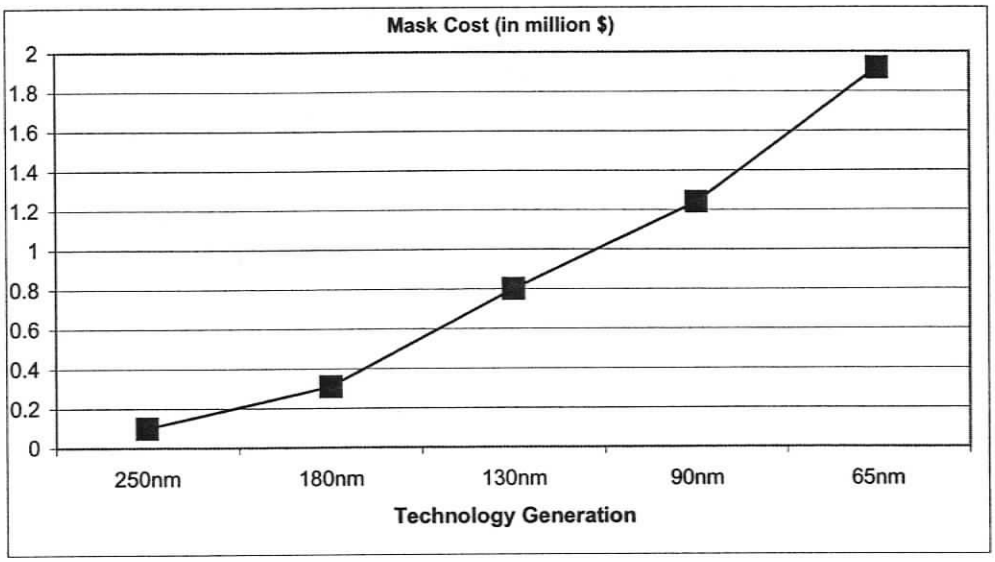

Figure 1.2: Increasing Mask Costs for Standard Cell CMOS.
Adapted from [10].

Cadence, the maker of a popular suite of Integrated Circuit (IC) design tools, define four categories of platforms. An article [11] written by Martin and Schirrmeister itemizes:

1. **Full Application Platforms** consist of libraries of hardware and software modules, from which a designer can choose and assemble a system. A versatile and comprehensive software API facilitates the remainder of the customization. Texas Instruments OMAP [12], ARM PrimeXsys [8], and Intel Xscale [13] fall into this category.
2. **Processor Centric Platforms** are based on specific processors and the software development flow. Custom hardware is often required to complete the project. Freescale's PowerPC [14] and MIPS Technologies' MIPS64 [15] are two processor families commonly used as bases for a processor centric platform.
3. **Communication Centric Platforms** are based around a communication fabric for the components of the system. Any module designed to be compliant with the communications bus can be easily reused in other systems. IBM's CoreConnect [16] and ARM's AMBA bus [17] are the most common examples.

4. **Fully Programmable Platforms** are based on flexible logic devices such as Field Programmable Gate Arrays (FPGAs) into which hardware modules can be instantiated. FPGA makers are offering reusable IP libraries, including processors optimized for the logic fabric. The most popular examples are Altera Stratix [18] and Xilinx Virtex [19] devices.

This categorization is useful for differentiating between platform vendors, based on deliverables and the desired design method, and should be helpful to anyone familiar with hardware/software co-design techniques. For example, if it is the system developer's intention to implement the majority of the functionality in software, it is appropriate to choose a processor centric or full application platform; the CPU, development environment, and existing software libraries are sophisticated and reduce the total development cost. If a full application platform whose hardware resources closely match the requirements of the system can be found, this is likely to be the best overall solution, since the hardware costs are amortized over many more units that would otherwise be possible, and the software development is simplified by a strong development support from the platform vendor. If such a system cannot be identified, the extra hardware provided on a full application platform may be wasteful, prompting the SoC designers to use a processor centric platform and add the custom hardware themselves.

On the other hand, if the workload is to be partitioned amongst many different hardware elements, a fully programmable platform or a communication centric platform allow for much greater hardware tailoring. A fully programmable platform generally has a larger hardware IP library to select from, and is simpler than a communication centric platform to author completely custom IP on. However, FPGAs and other fully programmable platforms traditionally have very weak communication resources; if complicated bus sharing or high bandwidth is required by the application, it may make more sense to employ a communication centric platform.

There is a multitude of other ways to divide the field of available platforms for SoC development. The four categories discussed in the preceding paragraphs are just one

example. However, we have illustrated that an important corollary decision must be made during the selection of the platform for development: how much hardware customization is required to achieve the desired function and performance from the chosen platform? For the remainder of this thesis, we propose and concern ourselves with just one simple division of the SoC development platform space, hard or soft, based on the designers' need for hardware customization, as follows.

1.2.1 Hard Platforms

Hard Platforms follow the example of the COTS approach to embedded system design, only this time using specialized platforms (not quite "off the shelf") as the primary ingredient. The chip is completely pre-designed, verified, and packaged, and sold as is. In addition to amortizing the development cost over multiple projects, hard platforms amortize the manufacturing cost over many more units as well, because the platform is manufactured to one all-encompassing hardware specification and is not subject to the customizations found in other types of platforms. Compilers, Integrated Development Environments (IDEs), prewritten software (especially operating systems) and an assortment of external, discrete components which have been successfully used in conjunction with the SoC are all part of the platform.

Because hardened platforms allow little to no customization of hardware, product differentiation is achieved in software. This model is best suited to standards-based industries such as cellular telephony, where several producers are attempting to solve the same problem. Texas Instruments' OMAP platforms [12] are aimed at cellular phones with additional features designed to complement multimedia and security. Intel's XScale product line [13] is another example of a hard platform.

1.2.2 Soft Platforms

Soft platforms are what most people refer to when they speak of designing with IP reuse. They allow for the instantiation of IP blocks as well as the addition of custom blocks and glue logic, and are typically designed with a Hardware Description Language

(HDL) or Computer Aided Design (CAD) tools. This is a significant Non-Recurring Engineering (NRE) cost, which must be amortized over the total number of units produced.

Soft platforms are delivered on one of three types of custom logic fabrics:

1. **Custom Application Specific Integrated Circuits (ASICs)** are fabricated completely from scratch, and are the most expensive to produce. Taking advantage of the latest fabrication technologies and the high degree of customization makes full-custom ASICs the highest performance option of any SoC. We include chips based on standard cell libraries in this category, as well as the large majority of mixed signal designs.
2. **Field Programmable Gate Arrays (FPGAs)** are SRAM (or more recently, FLASH) memory based devices which are pre-fabricated and bought off the shelf. They contain logic, routing, memory, and special-function resources (such as multipliers) which can be enabled and configured "in the field" to instantiate custom designs. Due to the overhead of all the resources which go unused when the customer's design is instantiated, these are comparatively inefficient in terms of size, power, and speed, but have the tremendous benefit of being re-configurable, making FPGAs an excellent choice for development and prototyping.
3. **Structured ASICs** are a compromise between the two fabrics above, and the most recent option on the market. A structured ASIC is composed of a matrix of transistors with clock trees, power nets, memories, and other simple functional blocks. As with FPGAs, the silicon is completely pre-fabricated, but a couple of metal layers are customizable to produce custom logic, or instantiate firm cores.

The choice between these three fabrics is mainly a tradeoff between cost and performance. All three fabrics incur the design NRE costs, though the vendors of each offer tools and resources to assist their customers in lowering these costs. At low volumes, FPGAs are the least expensive because implementation of a design incurs no

physical NRE; the chips are simply bought off the shelf and configured in the field. However, their relative physical inefficiency, and the production and validation costs expended by the vendor means a high cost per chip. ASICs, both structured and custom, require an initial cost outlay to set up a fabrication facility. This is a one-time cost, though, and once set up, the individual cost of each chip is low.

Volume is not the only factor. The more application specific the silicon (full-custom, structured, and FPGA, in descending order), the more efficient and powerful the end product. In extreme cases, a customer may require a higher clock speed, or less power consumption than is available with an FPGA, or even a structured ASIC, and would therefore be obliged to use a more expensive process.

1.3 Thesis Motivation and Outline

Simply put, the decision between using hard or soft platforms for SoC development is one of how much hardware customization is required. It is possible that an existing hard platform has a hardware profile which very closely matches the requirements of the intended SoC, in which case that platform will undoubtedly be the best choice for implementation. However, in the event that no such platform exists, the designers must choose between the inefficiency of a poorly matched hard platform and the cost of developing a soft platform.

In the first case, the designer is required to correctly evaluate the inefficiencies inherent in the chosen platform. On the other hand, if the decision is made to tailor a soft platform for the requirements of the application, the designers must decide between architectural options in order to optimize the hardware for the target workload [11]. In [2], Wolf states "we can't design an architecture until we understand the components' characteristics." The discussion and examples in this thesis are intended to assist hardware engineers in these two tasks:

- The evaluation of available hard and soft platforms,

- The optimization, where possible, of the hardware platform.

Similar applications will be affected similarly by such architectural decisions. Therefore, it will be useful to classify applications based on workload, and assign architectural optimizations useful to each class. Such a classification of applications is presented in Chapter 2 of this thesis. Chapter 3 follows, with a taxonomy of architectural features and enhancements available to the embedded designer, and explain the tradeoff decisions involved in employing each one. We focus our attention on optimizations relating to the efficiency of processors, rather than the system peripherals, because every platform contains a processor in order to support a range of different applications [20]. The various peripherals, and the potential of each to be optimized in their own right would dilute the discussion and prevent any reasonable conclusions from being reached. In general, the specifics of the processor and its interaction with the rest of the system, including the execution of software, have the largest effect on overall performance [2],[20].

In [20], Sangiovanni-Vincentelli says "Only by taking a high-level view of the problem can we devise solutions that will have a real impact on embedded-systems design. The essential issue to resolve is the link between functionality and programmable platforms." Together, Chapters 2 and 3 present just such a high-level view and link the functional requirements to individual architectural optimizations of the programmable portion of an arbitrary platform. Then, to illustrate this link we present two case studies in Chapters 4 and 5, from two different perspectives. The first illustrates the analysis and selection of architecture to enhance a single application, namely networking. The discussion and process in Chapter 4 (though not necessarily the conclusions) can and should be applied to hardware for any other application.

Chapter 5 goes into greater depth on a single hardware tradeoff, that of scratchpad memory versus cache, and the effects of the tradeoff on a multitude of application types. After critical analysis (such as in Chapter 4) yields a prioritized list of hardware issues to be studied, the most important ones should be empirically analyzed in the manner of

Chapter 5. Our presentation of the scratchpad versus cache issue is straightforward and accessible to a hardware engineer. As we show, however, the benefits of such an analysis, and the effect it can have on the final design, are significant. Finally, Chapter 6 presents our conclusions and suggest directions for future work.

2. Processing Requirements for Embedded Applications

The goal of this chapter is to explore the work performed by embedded computers at a fundamental level, as a means to better understand the hardware that satisfies those workloads. This is not merely a background survey; it is an original and analytical treatment of the matter which is a necessary foundation for the remainder of the thesis. We take a bottom-up approach, beginning with the most elementary requirements of embedded computing in Section 2.1, and the most basic units of embedded computation in Section 2.2. Finally, we use the framework in those two sections to describe from a high-level perspective, families of embedded workloads and their computational requirements, in the final section of this chapter.

We intend to provide a comprehensive survey and analysis. However, in the interest of brevity and a meaningful foundation for comparison later in this thesis, we omit from discussion two categories that may overlap with embedded computing, using the definition in Chapter 1.

- **Supercomputing** – Computing on its largest scale involves parallel processing by hundreds or thousands of individual processors, or processing with extremely powerful custom architectures. While some such systems do have the property of a fully defined application at design time, many do not, and the distinction is difficult to make on an architectural level. It is difficult to include such systems in a discussion of embedded computers in any meaningful way.
- **Microcontrollers** – The vast majority of computing in the world is performed by small-word processors (often called "microcontrollers") whose native data type is smaller than 16 binary digits [21]. As a rule, the simplicity of the applications usually handled by such machines makes them less interesting than the more complex applications handled by larger-word processors.

The remaining applications are diverse, but may be reasonably compared on similar architectures; by highlighting the sometimes subtle differences in applications while restricting the architecture to a few similar classes, we will show the need for architectural variations and optimization. We begin by laying the foundation for such comparisons.

2.1 Universal Design Constraints

The issues of power, complexity, and performance are considered in every design and with most decisions. The addition or enhancement of almost any feature comes at some cost to at least one of those three factors, and improves at least one of the others.

2.1.1 Performance and Efficiency

In this thesis, we make use of the terms *performance* and *efficiency* to describe the quality of some metric or measurement against some other measurement of cost. Unless explicitly modified (e.g., *energy* efficiency, *battery* performance), we define *efficiency* to mean the amount of work done (i.e., the proportion of the whole program executed) toward the application's goal, per unit of effort. Within this definition, *effort* is measured in terms of either energy or time consumed; in most cases, there will be a linear relationship between time and energy. When the relationship is not so simple, as when an improvement can be made which directly sacrifices time efficiency for power efficiency, or vice versa, we will be explicit as to which efficiency we refer. We define *performance* to be the rate at which that work is done, without respect to the cost. Therefore, one system may be higher-performance than another, but not necessarily more efficient.

Every system has a requirement for achieving a certain amount of work in a certain timeframe. In the case of real-time systems, this performance requirement is the defining attribute of the entire system. For other systems, performance may be a soft requirement; a "faster-is-better" goal for arriving at a solution, or performing a task. In the arena of general-purpose computing, faster is better in nearly every instance. This is not

necessarily the case in embedded computing, where performance requirements should not be treated as minimums but as targets. Over-engineering too much performance into an embedded design is a pitfall that can cause unnecessary difficulty in meeting the other two constraints.

In many embedded systems, power is not a major limitation. The computer systems found in cars, set-top boxes, toasters, etc. have effectively unlimited power, since battery life is not a concern. (They may not have unlimited cooling, in which case power dissipation must still be carefully considered.) On the other hand, in the increasing number of mobile embedded SoCs that have limited power budgets, battery capacity is not increasing as fast as power requirements are [22]. Observed improvements in modern battery lifetime are more readily attributable to exponential advances in integration and feature size, and architectural improvements such as clock gating, power stepping, etc., than to enhancements in battery technology.

There are some embedded computers that are constrained by the size of the end product, e.g., an MP3 player keychain, which would have to fit into a form-factor of only a few centimeters. In this case, it may be necessary to choose the most miniaturized and the most expensive manufacturing process. Power consumption is a factor in this as well, because cooling is greatly affected by the chip package [23].

2.1.2 Complexity and Transistor Count

As a design increases in complexity, through the addition of more features, or through increased sophistication of current circuits, it acquires more transistors. Each additional transistor imposes a system cost by demanding more energy when switching (active) and leaking more current even when inactive. Manufacturing costs increase as the designs consume more real estate on the silicon die, and as they increase the likelihood of a manufacturing defect. As well, the nonrecurring engineering cost of the design increases because increased complexity requires nonlinearly greater effort to design and validate.

Large chips have additional problems not related to transistor count, as well. Pin count tends to increase with complexity, and it can increase the necessary size of the die

or package independently of the transistor count. Clock skew from one side of the device to the other can be a problematic complication in high-speed designs, and is exacerbated by increased size and routing density of a design. In the best case, this only increases the difficulty of floorplanning the design, but in the worst case it can decrease the maximum attainable clock frequency and limit the performance of the device.

2.2 Functional Demands

In this section, we itemize the elementary units of work that a processor will be called upon to perform. From another perspective, these are the simplest operations a processor may or may not be capable of performing. We categorize them as in Figure 2.1.

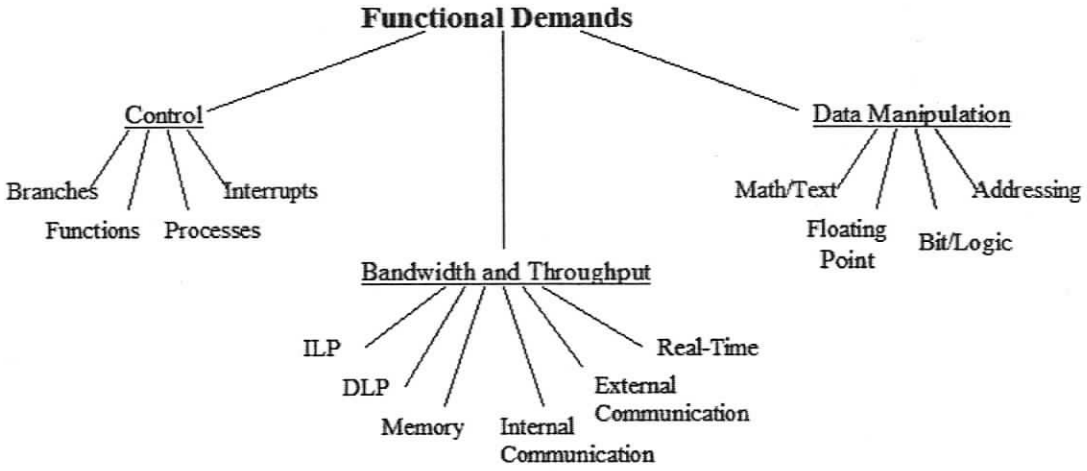

Figure 2.1: Classification of Functional Demands

2.2.1 Control

A group of instructions with only one possible execution path is called a *basic block*, and is usually only a few instructions long. Any longer portions of software can be expected to have multiple possible execution paths, and the exact path taken by the processor on any given iteration depends on data and external stimuli. The instructions that determine the taken path are control instructions, the arrangement and frequency of which is a fundamental property of each family of workloads. Text processing applications, for example, are heavily decision-oriented, and will execute a large proportion of conditional

and unconditional branches. (The EEMBC Text Processing benchmark is 35% compare/branch [24]; contrast that with an average of 16% conditional branch and 13% integer compare in SPECint92, a comprehensive integer processing suite [25].) Other embedded softwares, including most video algorithms, are highly loop-oriented. We will examine the different types of control constructs in this section, in order of increasing abstraction level.

2.2.1.1 Branches

A *branch* and its condition can usually be evaluated very quickly, and do not by themselves place a large burden on the processor. However, the effect of changing the location of current code execution can be detrimental to some of the measures undertaken to make the processor more efficient. Caches, which operate on the principle of locality, are difficult to keep populated with useful instructions and data if the location of the code is uncertain. Deep pipelines, in both the instruction path and specialized arithmetic or memory access units, can be interrupted by a sudden and unpredictable change in the code path.

The distance, frequency, and type of branch instructions within a piece of software are heavily correlated to the application of the software [26],[27]. For example, Figure 3.31 in [25] illustrates the marked difference in branch frequencies between integer and floating-point applications. Table 2.1 shows that, with three different compilers for the ARM Thumb® Instruction Set Architecture (ISA), similar benchmarks from the SPEC CPU benchmark suites have closely corresponding basic block lengths.

		A	B	C
175.vpr	FPGA Place and Route	2.91	2.78	2.83
300.twolf	LSI Place and Route	3.17	3.01	3.33
147.vortex	Object Oriented Database	2.68	3.24	2.87
255.vortex	Object Oriented Database	2.71	3.24	2.87
126.gcc	SPARC C Compiler	1.82	2.06	2.22
176.gcc	88100 C Compiler	1.82	2.04	2.19

Table 2.1: Basic block length, in instructions, of similar SPEC CPU benchmarks on three different compilers: A, B, and C.
Similar benchmarks are grouped and shaded similarly. Data from [28].

The code contained between a backwards branch and its target, i.e., a *loop*, is often repeated a large number of times. Some embedded applications (e.g., autocorrelation and Fourier transform in signal processing, especially in telecommunications) spend the majority of their execution time in relatively small loops, iterating a short task over a large quantity of data [24]. This special use of conditional branches is so commonly employed that, as we will see in Chapter 3, several hardware optimizations are available specifically to facilitate the efficient execution of loops.

2.2.1.2 Functions

The next level of abstraction in the software control family is *functions*. Function calls are implemented differently by each programming language, but always amount to a change in software flow at the processor level. From a machine-language perspective, function calls are difficult to distinguish from other unconditional branches. At an application level, though, they impose some overhead by occupying more runtime memory and passing arguments.

2.2.1.3 Processes

Moving up in abstraction, the notion of a *process* or *thread* covers the complete execution of a single stream of software. Most processors execute only one process at a time, but may switch between processes if the application demands. This process switching, or multitasking, is usually implemented in software by another process, the Operating System (OS), which can be as simple as a few instructions for performing the switch, or as complicated as a whole user interface and environment. The sophistication of the operating system depends heavily on how much the individual processes know about each other, and how the system resources are shared between them.

2.2.1.4 Interrupts

So far, we have only concerned ourselves with the execution of code as a measure of processor work. In many applications, however, control is also exerted on the processor by hardwired stimuli which trigger *interrupts*. (Software interrupts are more commonly called *traps*.) The implementation of interrupts is handled differently by each processor family, but is usually similar to a function call, albeit an externally triggered one. Real-time embedded applications such as industrial control and networking are greatly affected by the manner in which timer interrupts are handled. Instrumentation applications that rely on external (to the processor) sensors and transducers are similarly affected by the handling of non-timer interrupts.

2.2.2 Bandwidth and Throughput

Throughput is a measure of quantity per unit time. *Bandwidth* is a measure of a system's capacity for throughput, though in many systems the terms are used interchangeably. When discussing communications systems, bandwidth is commonly used to refer to both the capacity of the channel and the actual amount of traffic, even though throughput might be a more accurate term for the latter.

When discussing computers, throughput generally refers to a measurement that can be reduced down to a number of bytes processed by the computer in a given amount of time. Depending on the application, those bytes may comprise a data flow (as in video systems), a number of calculations (in cryptographic systems), or a number of instructions or program work processed. In any case, throughput is the true measure of speed performance for a computer, despite the attention given to clock frequency or other artificial metrics. The commonly quoted Millions of Instructions Per Second (MIPS) is a throughput measurement, though famously flawed [29]. Rather than measuring the rate at which a processor completes a program, or produces output, MIPS measures the number of machine-level instructions performed by the computer per second, and ignores the fact that these instructions are not absolute units of work across different ISAs. Still, if that caveat is properly taken into account, MIPS is a useful measure.

Many aspects of architecture can affect the system's bandwidth, or capacity to achieve results in a timely manner. For greatest efficiency, the entire system must be tuned to work in concert, so that no portion is retarded by a lack of performance from some other subsystem.

2.2.2.1 Instruction-Level Parallelism

Instruction-Level Parallelism (ILP) is the amount of mutual independence in neighboring assembly instructions, such that the processor may actually perform them simultaneously and in parallel. In a linear flow which does not exploit ILP, an instruction is exercising one functional unit while the other units of the processor are idle (for example, the branch unit does no work while the processor is carrying out a floating-point multiplication.) Having more than one instruction "in flight" at a time, by issuing more than one per clock cycle (*superscalarity*), or by issuing subsequent instructions before the first ones complete (*pipelining*), a processor can make more effective use of its resources. Figure 2.2a depicts a single-issue pipeline; notice that multiple instructions (1,2,3) are being processed at once. Similarly, Figure 2.2b shows a 2-way superscalar pipeline.

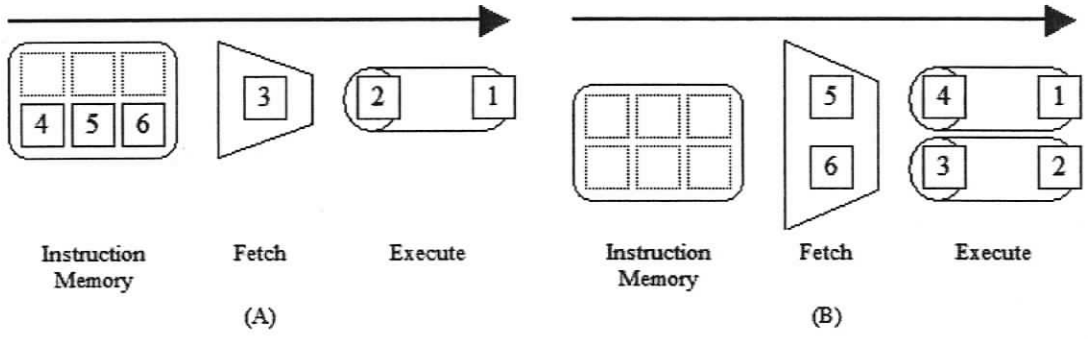

Figure 2.2: (a) Single-Issue and (b) Multiple-Issue Pipelines

ILP is a property of software, and therefore of application. Workloads with heavy control (as in Section 2.2.1) demands are particularly difficult to extract ILP from, as the frequent branches and jumps make it difficult to determine which instructions are appropriate to execute at any given time.

Both superscalarity and pipelining add complexity to a processor, and require significant logic resources that increase the design effort and manufacturing cost, as well as the size and power consumption of the chip. Heavy use of either technique is less common in embedded computing than in other forms, because of the considerable cost.

2.2.2.2 Data-Level Parallelism

Another form of parallelism is *data-level parallelism*, wherein the same operation is to be carried out on multiple data sets simultaneously. If multiple functional blocks are available, vectorization techniques (also known as Single Instruction stream, Multiple Data stream, or SIMD **[30]**) can be applied to the data in order to speed up processing. Many DSP, video, audio, and control applications exhibit data-level parallelism. Consequently, DSPs and other processors aimed at those markets are capable of very effective exploitation of data-level parallelism.

2.2.2.3 Memory Hierarchy

From the desktop computing world, we are accustomed to thinking of memory from a performance point of view; 512 MB of RAM makes for a livelier PC than does 256, because more applications can be served by the larger RAM simultaneously, and the operating system does not have to access the swap file on the hard drive as frequently. We are also aware of a minimum memory requirement below which a program simply cannot be executed. On what is usually a smaller scale, these issues exist for embedded systems as well, but with a different implication; in general, increasing performance beyond some minimum threshold only serves to increase the cost of the design, with diminishing return.

In many systems, the impact of memory on the overall performance is more dependant on bandwidth and latency than overall memory size. High latency caused by inappropriate architecture restricts the ability of the processor to operate efficiently when it must stall awaiting the next instruction or datum. Choices made when selecting the memory technology, the arrangement, associativity, and replacement rules for caching,

and the number of hierarchical levels, amongst other things, affect the latency of individual or block memory accesses, which is critical in many embedded applications.

2.2.2.4 Internal Communications

The capacity of the bus connecting the processor to the memory hierarchy can limit SoC performance as well. There are several factors to consider: total available capacity is the most obvious, but it is also necessary to factor in the overhead (both time and capacity) inherent in bus transactions and control, and the mating of bus and processor clock frequencies, so as not to create artificial penalties involved with handshaking.

In fact, depending on the application, peripheral devices such as network ports, coprocessors (possibly including a GPU), and A/D or D/A converters are affected similarly by bus limitations. Any application that demands the movement of large amounts of data around the SoC will need to pay careful attention to the bus architecture.

There is a wide selection of bus standards for in-system communications. Some, such as AMBA [31], or CoreConnect [32], were created by specific vendors (ARM and IBM, in these cases) for the improvement of their own products; though the standards are widely disseminated in order to facilitate their use. Other bus specifications are industry standards, such as PCI [33], and have been developed with specific applications in mind, and so are optimized for those types of data traffic.

2.2.2.5 External Communications

There are many competing and coexisting standards regarding how a SoC communicates with external computing systems. Ethernet, Firewire, USB, Bluetooth, etc. are designed for different requirements, and the choice to adopt a standard depends on many factors, including whether the system is wireless or not, and how much bandwidth is required. In general, the bandwidths involved for external communications are much less than those required on internal busses (as in Section 2.2.2.4). There are exceptions, however, as with systems utilizing Gigabit Ethernet or SONET. It should also be understood that, unlike in internal communications, the organization of data into packets or other transfer

units and the overhead required (much of it in software) to implement the specific protocol could place a significant demand on system resources. It takes roughly the processing power of a 1GHz Pentium 3 to process 1Gb/s of TCP/IP data [34]. Integer mathematics and bit manipulation are essential for these tasks.

An important consideration is how responsive the system must be to data requests. For example, the EEMBC includes "CAN Remote Data Request" [35] as a metric in its AutoMark test suite, indicating the importance of external communications in automotive processor architectures. Similarly, EEMBC's NetMark [24] is heavily based on packet throughput scores, for networking devices whose sole purpose is to direct, transform, or otherwise transport data. The existence of benchmarks like these is testament to the importance of reliable and efficient external communications in embedded systems. Because so many embedded computers exist to supply data to or react to data from a larger system, this is an essential concern for the system designer.

2.2.2.6 Real-Time

"Hard" real-time constraints are those for which an unmet deadline causes a system fault, as in control systems. "Soft" real-time constraints are those for which missed deadlines cause a drop in quality, such as video display. SoCs operating under real-time constraints must concern themselves with controlling the maximum latency of any given task. Execution time of the task itself is obviously important, and therefore so is the overall processing speed of the system. However, latencies associated with external event recognition, task switching, and resource sharing are critical as well, and often more complicated. When specifying the system, one must be able to establish maximum bounds for the time until completion of each real-time task. In general, the simpler the hardware, the easier it is to accurately estimate those bounds. However, as we will see in Chapter 3, there are certain hardware features that directly affect the compatibility of the system to real-time workloads.

2.2.3 Data Manipulation

This section is closely related to throughput. However, the ability to perform individual operations on data is a separate concern from the ingress and egress of that data. This section concerns the handling of data and information within the processor, and the efficiency of the individual functional units. Optimizations and features that improve one requirement do not directly improve the other, though the effectiveness of the SoC as a whole relies on performance in both aspects.

Generally speaking, the functional units that perform the work in this class make up the majority of the processor, because each operation type requires its own specific hardware to execute. In fact, in processors designed to exploit either instruction- or data-level parallelism, there may be multiple copies of some or all functional units.

2.2.3.1 Simple Mathematics / Text

This is a broad group of simple operations on a variety of simple data types. The operations involved, such as add/subtract, multiply, divide, and compare, require relatively simple hardware. (We are referring to straightforward multiplications and divisions here, not the more exotic operations found in specialized math processors and DSPs.) Embedded applications with a heavy proportion of these operations tend to benefit from instruction- and data-level parallelism. As a result, there are often multiple instances of the required, often pipelined, functional units.

For integers, the arithmetic units may either accommodate signed or unsigned data. Separate units are also required for each supported fixed-point or floating-point type. Finally, additional hardware may be required if short (e.g., 4- or 8-bit) and long (e.g., 64- or 128-bit commonly, though others exist) data types[1] are to be supported.

Text is usually represented as integer codes (e.g., ASCII or Unicode). Most text processing is done by comparison operators, although simple addition and subtraction is

---

[1] The categorization of *short* and *long* data types assumes a processor with native 32- or 16-bit word lengths. The majority of this thesis deals with such processors; we defend that decision early in Chapter 3.

frequently required to manipulate position counters and pointers. Applications which deal with low-precision decimal numbers (for example, monetary values in a $d.cc format) are most easily implemented with simple integer units and decimal-shifted after calculation, rather than in bulky and complicated floating- or fixed-point units.

2.2.3.2 Floating Point Arithmetic

Integer mathematics are simple, but limited in magnitude and therefore application. In order to capture a wider range of numbers without increasing the bit width to an unmanageable size, a floating point data type must be employed. However, in order to operate on floating point data, the hardware must undergo several extra steps not required for fixed point or integer operations. The additional hardware and/or latency required can be significant. The most common forms for floating point data are the IEEE 754 standard [36] single- and double-precision formats which requires 32 and 64 bits to represent, respectively.

A special category of floating point workload is three-dimensional graphics, a computationally intensive task whose demands for state-of-the-art quality have evolved to require highly specialized hardware structures and communications. This category deserves special notice because of the overwhelming amount of effort that has gone into producing specialized hardware for the one explicit purpose. In fact, in some systems, the computational effort required to produce 3D graphics outweighs the rest of the system demands. For example, the latest Nvidia Graphics Processing Unit (GPU) has 220 million transistors [37] as compared to 178 million on the latest Pentium 4 EE [38].

Rendering 3D graphics requires complicated manipulations of large buffers of values in order to perform lighting, shading, texture mapping, antialiasing, and rasterizing, amongst other operations, often referred to as the *graphics pipeline*. The operations involved are repeated many times in order to arrive at each pixel, and multiple levels of parallelism are available to be exploited.

2.2.3.3 Bit Manipulation and Logic

Many embedded applications operate on data in non-standard formats, which do not fit neatly into bytes or machine words. For example, several binary signals may be sampled simultaneously and concatenated into a single word. Control registers within the processor itself often contain such aggregated values. Certain types of data streams, for example, packets or cells in telecommunications standards, pack adjacent values into unwieldy fields within the stream. To operate on such data types efficiently, a processor can benefit from a flexible and varied collection of bit-masked operations, including all the Boolean and mathematical logic operators. The presence of hardware to perform these operations can save critical cycles which would otherwise be spent copying, manipulating, and reconstructing data or control signals.

2.2.3.4 Addressing and Pointer Arithmetic

To support the assembly language programmer and compiler, and accommodate a variety of high-level data structures in order to simplify algorithms, a wide variety of addressing modes have been explored. Depending on where the data is stored, where the address is stored and calculated, how the high-level data structure is organized, and how the physical memory is accessed, any of a very wide variety of addressing modes might be most appropriate. Multiple addressing and reference formats each require additional hardware to translate, and that hardware must be optimized to complete as quickly as possible, since addressing is most commonly a secondary effect of the intended operation.

Table lookups, stack operations, tree traversals, and a number of other common operations based on pointer arithmetic comprise a large portion of some program types. In these cases, efficient manipulation of the pointer to memory, usually held in a register of some sort, is essential to optimal processing. Pre- or post-increment or -decrement operations are useful, for the full variety of data sizes (1-, 2- and 4- bytes being the most common). So too are scaled or indexed addressing modes. All of these operations can be carried out on the pointer by the ordinary integer arithmetic functional units, but doing so

requires additional instructions and code which can make up a very large portion of the total run time. It is therefore common for a processor to contain specialized address arithmetic units.

2.3 Classification of Workloads

The foregoing elemental demands are combined in a variety of ways in the workloads of embedded SoCs. In this section, we present a classification of high-level embedded workloads. To illuminate the discussion, we provide common examples of real world embedded products which exhibit these workloads. The paragraphs to come will detail the classification in Figure 2.3.

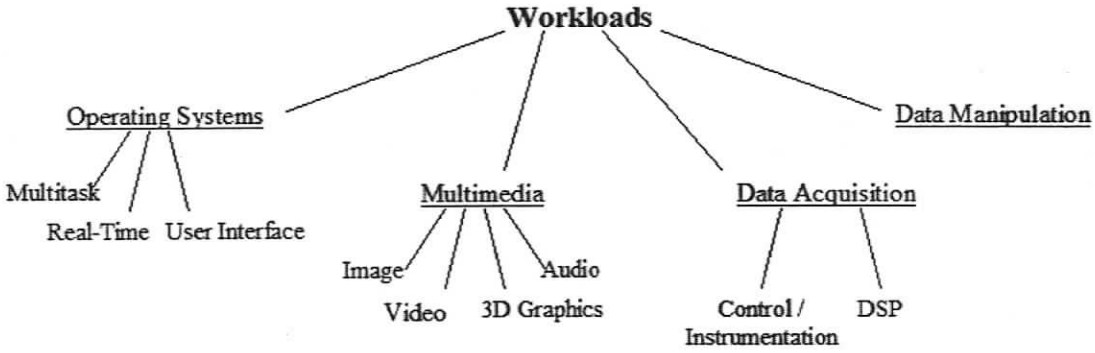

Figure 2.3: Classification of Embedded Workloads

2.3.1 Operating Systems

The more complicated a system, the higher the demand for an operating system. An OS greatly simplifies the programming model and performs vital low-level control and maintenance, as discussed below.

2.3.1.1 Multitasking

The OS is responsible for the management of all resources, including the processor itself, which means that the OS is responsible for switching between active tasks to ensure that

each receives sufficient processor time to accomplish its goals. The OS also manages system peripherals such as memory and communications ports.

Multitasking is desirable in any system that must perform multiple separate functions. In the embedded world this property is found in user-devices like Personal Digital Assistants, which like desktop computers must perform their computations and respond to user (and network) input simultaneously. It is also found in many controller applications, as in industrial production or in avionics, where multiple data streams are monitored independently by a single computer system.

2.3.1.2 Real-Time

One of the key resources that might be managed by an OS is the Real Time Clock (RTC). The example of a PDA above is not necessarily a real-time system (even though timely reaction to user input is highly desirable), but the industrial production example almost certainly is. A real-time OS is responsible for ensuring that, when managing multiple tasks as above, the tasks are guaranteed to meet certain deadlines. In some cases, as with certain video applications, a real-time OS may not even be required to manage multiple tasks, but just ensure that a single task is synchronized to its deadlines and inputs.

2.3.1.3 User-Interface

Not all systems that require operating systems have direct human user control. Automotive computers are rarely accessed by users, for example. Those that do must often provide the user with an interface environment, usually graphical, from which to receive data and issue commands to the system. The actual presentation of that data and reception of those commands may be treated as any other tasks, with attending real-time requirements (or not). The most obvious embedded examples are PDAs, but the menu systems of DVD players and the combination of keypad and display for point-of-sale terminals are also examples of user-interface supplied by the OS.

2.3.2 Multimedia

2.3.2.1 Still Image

Presentation of still images is the simplest multimedia workload, in part because there is no intrinsic time requirement, aside from the obvious sooner-is-better. Basic bitmaps require only the manipulation of pixels on a display, a task that has modest integer requirements and simple bit-field manipulations, because several individual pixels are usually concatenated (as in the industry standard RGB or YUV formats) to form a single machine word. Many simple displays, such as those of GPS handhelds, or simple Graphical User Interfaces (GUIs) are adequately served by such tasks. Compression, and decompression of raw images into GIF- or JPEG-type formats (the two most common compression types), as required by digital cameras, or noise reduction and image enhancement filters require significant buffering resources and pointer manipulation.

2.3.2.2 Video

Video applications are plentiful in the embedded marketplace. DVD and HDTV make up the majority of this segment, but cellular phones, digital cameras, PDAs, and surveillance products have all got video requirements as well. Simple video consists of a series of time-linear frames of pixels. Since it requires a great deal of bandwidth to provide video in such a format, it is common for video to be compressed using a codec like MPEG-2. DVD and HDTV both use MPEG-2 encoding, which can have very high bandwidth requirements. Full-sized HDTV requires as much as 19 Mb/s of data [39]. Lower-bandwidth codecs intended for wireless applications, such as H.264, have similar processing requirements.

MPEG-2 consists of a stream of packets, each of which has a header to indicate whether it contains audio, video, or control data. A decoding system must first multiplex these packets, then handle them accordingly. Video packets are further processed by passing through an Inverse Discrete Cosine Transform (IDCT), and a Luminance-Chrominance (YUV) to RGB filter, both of which require many integer multiplications and additions. In order to extract the data from the bitstream, however, and perform the

Variable-Length Decoding (VLD) many irregularly-sized bit manipulations must be performed. [39] This is a key area in which custom hardware can be many times more efficient than the regular functional units in common processors.

Finally, synchronization of video to audio, and jitter-free presentation of both to the viewer is greatly benefited by a real-time OS.

2.3.2.3 Three-Dimensional Graphics

The rendering of 3D graphics is quite different from two-dimensional video. Characterized by complex algorithms in floating-point arithmetic, and high levels of data-level parallelism, owing to the repetitive nature of operations on multiple data points, rendered graphics have extremely high workloads. The latest graphics cards (which can be considered an 'embedded' system within the larger framework of desktop computers) and set-top game consoles have more transistors than even the most sophisticated CPUs, in part because of the very wide data paths.

2.3.2.4 Audio

Audio is similar to video in two respects; both have an intrinsic real-time requirement which defines the quality of their presentation, and both are fairly simple in raw-data formats, but most often require compression or decompression in real-world applications. Audio differs in that it consists of a stream of single data points, instead of being arranged in frames, and that the word lengths are larger to support greater sample precision. The types of operations involved, however, are similar. Of course, much lower bandwidths are involved, so devices such as portable music players and CD players have much smaller processing requirements, in comparison to video. In fact, it is more common for CD players to be implemented using discrete application-specific components [40],[41] rather than a processor-based system. (Additional individual components are employed for spindle- and laser-control, as well as Reed-Solomon decoding on the raw data stream.)

2.3.2.4.1 Voice

A special case of the audio workload is voice compression. It cannot be overlooked, however, because of the utility of efficient compression for cellular telephony, which may be the single most important embedded application of the 21st century. (Certainly it is the application driving the majority of advancements in the SoC field.) Voice, as an audio signal, has particular characteristics that allow it to be treated differently from music or ambient sound (as must be captured in television or cinema) [42]. However, even though the block-level operations are different for voice than other audio, the fundamental operations are still integer DSP, and the workload is similar.

2.3.3 Data Acquisition

The primary purpose of many embedded systems is the gathering or monitoring of data from sensors. The amount of processing performed on the collected data varies from zero, (e.g., inventory control, simple instrumentation) to light (e.g., flight data recording, compression) to heavy (e.g., motion estimation, voice encoding). In such systems, the bandwidth, format, and, timing of the data stream are the primary factors in defining the nature of the embedded processing system. We describe two categories, below.

2.3.3.1 Control Systems / Instrumentation

In the case of Supervisory Control And Data Acquisition (SCADA) systems, such as for remote oil well monitoring, or power line management, there is a user-interaction component to the system as well as simple acquisition. The system may be primarily responsible for recording and/or transmitting the acquired data, or it may involve some level of control of the manipulated system. In other, so called "deeply embedded" systems, user involvement is either nonexistent or very minimal, and the embedded computer is primarily responsible for reacting to gathered data. The majority of avionics and automotive computers fall into this category; traction control, valve timing, and pitch/yaw stabilization are prime examples.

The bandwidth and performance requirements of such systems vary greatly from end application to end application. Similarly, it is difficult to make generalizations about the types of data operations and memory needs of these systems. They do commonly share one aspect, however; nearly all have rigorous hard real-time constraints.

2.3.3.2 Digital Signal Processing

Digital Signal Processors (DSPs) are characterized by a steady stream of incoming data (often from external sensors) which must be heavily manipulated to produce an output data stream. Many multimedia applications fall into this category, especially those that call for alterations to the source data: audio equalization, video gamma correction, surround sound, and many others serve as examples. Other DSP applications include radar, Software Defined Radio (SDR), and cellular telephony.

DSPs may operate on integer, fixed-, or floating-point data, depending on the application. The algorithms involved tend to make heavy use of multiply-and-accumulate operations for convolutions, table lookups for trigonometric or other nonlinear operations, and indirect references for arrays and vectors of data points. Timely application of the algorithm is nearly always a factor, since DSP algorithms tend to operate on live sampled data.

2.3.4 Networking and Communications

Many embedded systems are defined by their ability to communicate with other computer systems, either for ingress or egress of data, or both. Some systems, such as BlackBerry text message terminals, Global Positioning Systems, or point-of-sale terminals have low bandwidth requirements. Other devices, like HDTV terminals, network routers, and external hard drives move very large amounts of data. Some devices demand wireless communication, some demand bi-directional or even multipoint communications. The application and its characteristic throughput and latency traits determine the protocol, topology, and physical medium for communications. These

topics are explored in depth, alongside SoC architectural features intended to optimize the fulfillment of the requirements, are discussed at length in Chapter 4.

2.4 Chapter Conclusions

Having reviewed and analyzed the processing requirements for embedded applications, we can conclude primarily that the variety is large. While there is some overlap in workloads across applications, there is also a great deal of diversification. It follows logically, then, that hardware should follow suit; different applications should require different hardware. In the next chapter, we examine the subject of hardware diversification as it relates to achieving greater efficiency in processing for different embedded applications.

3. Architectural Optimizations for Embedded Systems

In this chapter, we discuss individual architectural decisions available to the embedded system designer. We highlight the pros and cons of each feature in terms of the workloads and constraints discussed in Chapter 2, so that a system designer may make a more informed choice when selecting a processor for their specific project. Most of the optimizations visited herein are available in commercial chips and cores. The DSP processor market, in particular, is already highly optimized for certain classes of applications. On the other hand, many legacy general-purpose cores have a great deal of room for optimization, since they were not designed with modern requirements in mind, and in some cases were developed before some of the techniques were well-understood.

There are newer products, intended for specific application classes, which could benefit from further processor and/or system optimizations. For example, the Texas Instruments OMAP2 platform for cellular telephony includes an ARM11 processor [12]; the ARM11 is a very feature-rich processor designed for, according to the ARM11 core family website: automotive, data storage, imaging and embedded control, network infrastructure, consumer, and automotive infotainment **[43]**. Modern cellular phones have heavy voice, radio DSP, and video processing requirements. Assuming that the TI DSP included in OMAP2 will handle the majority of the radio requirements, and the dedicated graphics accelerator will handle the majority of video, it is likely that the many features of the ARM11 processor will be underutilized. One important reason for the analysis of hardware functionality is to avoid such wasteful design, and the costs incurred (at both design time and run time.)

This chapter begins with coarse details of available processors and processor cores then proceed to specifics. The particular tradeoffs of each aspect are discussed, with emphasis on their suitability for specific end-user applications.

3.1 Scope

When compiling the list of features discussed in this chapter, we sampled a large cross section of embedded 32-bit processors, both COTS products and IP cores, paying special attention to the most commercially popular ones. While it would be prohibitively difficult to capture every conceivable nuance of SoC architecture, we have made every effort to capture the most common and most important aspects. Particular attention is paid to those features whose inclusion or omission represent a direct tradeoff in performance between different classes of embedded application. That is, the most interesting discussion concerns the application-specific performance benefits of a particular hardware decision, especially when the same decision negatively affects other applications.

The topic was intentionally restricted to features of SoC architecture. External to that scope is any aspect of fabrication, packaging, cooling, voltage, or any other physical property not directly controlled by the architecture. Also omitted are external features such as communications busses or peripherals, except as their interface relates to SoC internals. Next, we refrain from discussing those features which are nearly universal, or whose variations from one architecture to another have little bearing on the performance of any particular application class. For example, nearly every major processor implements software trap instructions which facilitate, among other things, the arbitration of shared resources by an operating system. Variations in the particulars of such features are of negligible consequence in this discussion. Finally, we omit any discussion of hardware design-for-test because the tradeoffs involved in their implementation are not made solely within the context of end-user run-time performance.

3.2 Hardware Generalizations

In this first section we discuss high level topics applicable to *all* embedded systems. These are not features so much as design philosophies, and as such they directly and profoundly affect the hardware of the SoC.

3.2.1 RISC/CISC

No high-level discussion of computer architecture is complete without a debate over the relative merits of Reduced Instruction Set Computers (RISC) and Complex Instruction Set Computers (CISC). However, it is not the intention of this thesis to weigh in with an opinion or defense of one school or the other; we will merely make some observations pertinent to the topic of architectural tradeoffs in embedded computing. First, an explanation is in order.

As processors were evolving in the 1960s and 1970s, new instructions and capabilities were being added to processors in order to facilitate the production of more sophisticated software (and since compilers were so inefficient.) Memory accesses in particular were augmented by providing extra addressing modes, and/or the ability to directly modify memory using a single instruction. The high cost of memory in computer systems was a primary factor; it was desirable in some circumstances to have specialized instructions that performed complex operations which would otherwise take many smaller instructions to implement. This lowered the instruction count and the number of required accesses to instruction memory.

Beginning in the early 1980s, the focus shifted towards increasing the speed of the processor itself. Compilers were improving in their ability to efficiently manage hardware resources, but typically did not utilize the more complex instructions available to them – the very instructions that gave CISC machines their power. The idea that a simpler instruction set would be easier to support in compilers and simpler to streamline for performance in silicon gave rise to the RISC revolution. Efficient pipelining and increased reliance on the compiler to decode complex functions into simple instructions allowed processor performance to skyrocket throughout the 1980s and 1990s. These days, most modern architectures are primarily RISC ones, or have RISC roots. Even the x86 architectures, which many people cite as the last bastion and greatest success of the CISC approach, have actually become very RISC-like machines that just happen to support a legacy CISC instruction set. Comparatively minor attention has been paid to

optimizing the more "complex" instructions relative to the monumental effort in pipelining the most frequently used features.

Nowadays, processor speeds have completely outstripped memory speeds. This disparity along with the increasing energy cost of off-chip memory accesses means that memory is a major bottleneck again. Software is growing in complexity at an incredible rate; the software costs of most projects dwarf the hardware costs. The complexity of the software and the need for the most efficient code possible means that compilers are once again overburdened. These two factors, the motivations for the original CISC movement, are once again dominating factors in system performance.

Since the mid 1990s we have seen complex features creeping back into instructions sets [44]. The emergence of specialized architectures for specific markets (exactly what this thesis espouses) has meant the adoption of exotic specialized instructions, but not in the same manner as the original CISC approach. Instead, the instructions sets are carefully analyzed to ensure that only the most useful instructions are included, and the architectures are heavily optimized to support them. This is the RISC philosophy at work on CISC-like features.

In the world of embedded computers, many older CISC architectures are still gainfully employed. Motorola's (now Freescale's) 68K series is a perfect example. Exact numbers are difficult to come by, but one well-respected industry analyst, Jim Turley, reported that Freescale was still shipping nearly 75 million 68K chips annually, as recently as 2002 [45]. One reason for this is pure inertia; designers prefer to work with systems with which they are most familiar. Firmware programmers in particular often have many years invested in learning tricks and techniques applicable only to the complexities of their particular CISC processor of choice, and the cost of migrating to a new platform is not equal to the benefits. Once a company has tools and hardware IP (in the form of circuit schematics and preferred components) for one processor, it will find reasons to use them for any project they can stretch to. Another reason is that the individual features, on their own merit, may provide for more efficient processing, on an application-targeted basis, than the pure clock frequency-oriented performance of a

RISC. In section 3.3, we will be discussing the relative merits of some specific features, for the various applications outlined in Chapter 2.

3.2.1.1 Instruction Subsets

ARM's Thumb and MIPS' MIPS16e are extensions of the 32-bit native instruction sets on processors that support them [46], [47]. The 16-bit versions of the instructions are just shorthand, referring to a subset of the total register file and instruction count, and are translated into the full-length 32-bit instructions when processed. Other processor families, such as those produced by ARC International, freely mix 16 and 32-bit instructions without changing operating modes [48]. SuperH's SH-4 is a family of 32-bit processors with an entirely 16-bit instruction set [49]. ARM has recently released a second generation of Thumb, called Thumb-2, that is a significant departure from the philosophy of the original. Whereas Thumb was a second operating mode, Thumb-2 is complete instruction set containing both 32- and 16-bit instructions. ARM claims a 26% reduction in code size compared to 32-bit code, but 25% better performance than standard 16-bit ISAs with Thumb 2 [50]. However, the Thumb-2 code is likely to have larger size than a 16-bit ISA, and complete tasks more slowly than the standard 32-bit ARM ISA [51].

All of these abbreviated instruction sets are very much in line with the RISC philosophy of simpler instructions (despite having varying length in some cases) and less hardware. However, as a unique feature of the embedded design space, they differ in one major respect; the purpose of the hybrid instruction set is to accommodate and exploit applications which do not require heavy processing, and alleviate hardware (especially memory) burdens. The goal of traditional RISC methods was always to increase processing capability to meet increased demands, whereas the instruction subsets above intentionally decrease capabilities to match periods of decreased demand.

Any application which experience periods of low processing demand separated by periods of high demand, such as many user-interface applications, can benefit from the

lower power consumption and decreased storage requirement. Cellular phones are a perfect example, which is one reason that ARM (and Thumb) dominate the market.

3.2.2 Clock Frequency

For any given processor, increasing the clock frequency increases the amount of work it can do in a certain period of time. However, power consumption and generated heat increase linearly with clock frequency. Those factors alone should convince embedded systems designers to operate their processor at the minimum frequency that will accomplish the task.

Not all processors are created equally, though, and dissimilar processors can vary widely in their capabilities, even at the same clock frequency. The real-world CPI (Clocks Per Instruction) partially describes the clock efficiency of a computer and is characteristic of the processor, but also varies from workload to workload based on the mix of elementary functional demands (see Chapter 2). If an application is heavy on multiply-and-accumulate (MAC) instructions, but there is only one MAC unit in the ALU, the program must waste clock cycles while waiting for each MAC to complete. If the application has many branches, and the Branch Prediction Unit (BPU) is unsuccessful at predicting them, the CPI will fall due to branch penalties. For any given processor, increasing the clock frequency will decrease the effect of these hazards on instruction throughput per second. However, it is more efficient to ensure that the chosen processor has an appropriate allocation of hardware resources to eliminate or reduce the effect of hazards in the first place.

There are other architectural hazards which can not be mitigated by increasing the clock speed. If the processor is waiting for data from an external source, such as an A/D converter or a camera, no amount of increased processing power will help the computer meet its deadlines. Similarly, if the memory system lacks sufficient bandwidth to supply the processor with data or instructions at the rate required, the processor will be forced to stall while waiting. Once again, the demands of the application are the key determinant for system architecture; simply throwing more power at the problem does not solve it.

Throughout the rest of this chapter, we will endeavor to illustrate architectural solutions to application demands, as well as noting their general effect on the system clock frequency as well as CPI.

3.3 Specific Architectural Features

In this section, we describe some common features and architectural decisions found in modern embedded processors. For each feature, the tradeoffs with respect to the embedded system requirements in Chapter 2 are discussed. Features which might provide some benefit, but at negligible detriment to any other requirement are omitted from discussion. As mentioned previously, the addition of most features adversely affects complexity and power consumption while enhancing some aspect of performance. The more interesting ones are those that positively affect one or more requirements at the cost of some others.

3.3.1 Harvard / von Neumann Architectures

A Harvard machine is one which has separate paths to data and to instructions; the two are stored in logically distinct memories (even if the physical device is the same) and accessed via distinct busses. The Princeton, or von Neumann, approach is a unified memory bus upon which both instructions and data are transferred. The nature (unified or separate) of the address and data busses is a fundamental characteristic of any processor, and not a "feature" which can be added or modified easily. As such, it would be difficult to make this modification in any existing platform; if one or the other is required, the choice should be made early in the processor selection process. However, we include this characteristic in our list of architectural features in order to discuss the ramifications it may have on system performance and other aspects of architecture.

In general, the Harvard approach is capable of higher performance, if only because it has the potential for doubling the memory bandwidth; during each processor cycle, one instruction and one piece of data can be fetched simultaneously. The added performance

comes at the cost, however, of necessitating two separate busses which take up valuable routing resources on-chip, consume extra power, and require additional logic to arbitrate and control. When the busses are extended to off-chip memories, the pin count, chip size, and power consumption are increased dramatically. For this reason, many Harvard machines maintain separate physical busses for on-chip transactions, but combine them into one physical bus for off-chip access. There are other techniques and factors which can reduce the power consumption further, but the design space is complex and specialized and outside the scope of this thesis.

The on-chip memory hierarchies of the two systems can be quite different. Harvard architectures typically have separate instruction and data caches, while von Neumann ones must employ a unified cache. Separating the caches means losing some amount of flexibility at runtime, because it is impossible to determine exactly what the dynamic mix of cached instructions and data is optimal. However, separate caches do have the advantage of being separately configurable, so that the data cache can have different associativity, line-size, and replacement characteristics than the instruction cache. Since instructions and data usually have greatly differing access patterns, this can lead to an overall increase in the efficiency of the local memory structure. This efficiency advantage is likely to tip the scales in favor of Harvard for most embedded devices, especially when the memory requirements are high, as with video and DSP systems.

The von Neumann system is simpler, incurs less silicon and routing, and requires simpler memory structures. In some cases this simplicity can, in fact, lead to greater performance, especially if it allows higher clock speeds due to simpler logic paths. It does simplify the programming model somewhat, though that is a minor concern since compilers are adept at the sorts of considerations required by a Harvard approach. The only appreciable difference in the software capabilities of the two approaches is the possibility of von Neumann machines to run self-modifying code. Von Neumann systems are more appropriate for low-cost and low-energy devices, or in high-performance devices when the access patterns cannot be known at design time, as with PDAs.

3.3.2 External Memory

Every computer must have its program stored in some form of nonvolatile memory. In most embedded systems, the program is stored on a ROM, or, if persistent storage for generated data, a flash memory chip. Hard drives are becoming more popular as technology improves to miniaturize and ruggedize them, but the majority of embedded devices are still flash based [52]. In the next few sections, we explore the elements of external memory architecture as related to embedded systems.

3.3.2.1 External Working Memory

ROM or flash is necessary because it retains its contents even when the power is off. However, they are often unsuitable as the main operating memory in a system. The alternative is to include a DRAM device[1] which is immediately loaded with the full or partial contents of the nonvolatile memory upon system power up. Adding DRAM increases the component count and complexity of the system, but counters the disadvantages of flash, namely, slow and power consuming random memory access. DRAM, however, has the additional drawbacks of more complicated row-and-column addressing and continual refresh cycles.

The tradeoff here has nothing to do with the amount or type of memory required for storage purposes; an embedded system needs as much as it needs. (This point is discussed further in Section 3.3.2.2.) The question is really whether to run the program from nonvolatile memory, or from a working memory like DRAM that is quicker but more expensive. If more than some minimum access rate is required, it can make sense to add the DRAM in order to keep total energy cost down. Below that minimum, it is better to run from nonvolatile memory, provided the access latency is tolerable. Figure 3.1 illustrates the tradeoff curves using flash as an example of nonvolatile memory; it is representative of typical flash and DRAM access costs as taken from a sampling of data

---

[1] SRAM is generally reserved for cache, or other tightly coupled memory because of its higher cost and higher speed.

sheets. The exact crossing point depends on the particular characteristics of the flash and DRAM components used.

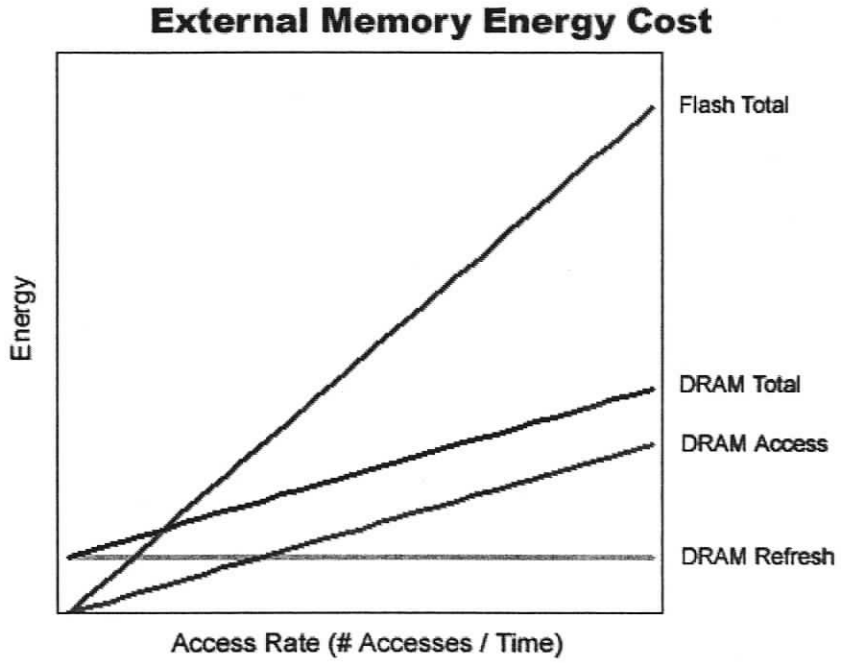

Figure 3.1: Flash vs. DRAM External Access Energy Cost

We can see that there is some (relatively low) access rate above which it is more effective to access DRAM than flash. Factoring in the cost of the additional DRAM component, logic or code for implementing refresh cycles, and slightly more complicated address decoding, and the system cost tradeoff point will move slightly in favor of a flash-only system, but for many systems, the inclusion of DRAM is appropriate. Only very simple embedded systems and those that are very sensitive to manufacturing cost, e.g., remote sensors, are likely to omit DRAM.

3.3.2.2 External Memory Size

With foreknowledge of the application, it is possible to design embedded systems in such a way that the memory is sufficient to hold the entire working set of all programs. The ISA and the compiler have large effects on code density [53], and can greatly affect the memory requirements, but such issues are easily resolved during the design phase. In a

system that includes RAM as main memory, accesses to nonvolatile secondary storage need only occur on startup to load the program into working memory or to store data in bulk. Increased main memory size is only desirable if it prevents frequent accesses to secondary storage. Any additional memory beyond that minimum is unnecessary and serves only to increase manufacturing cost. This is in distinct contrast to reprogrammable systems, such as desktop computers, which rely on large amounts of main memory to improve the performance of various programs (sometimes simultaneously) which are unknown at design time.

3.3.2.3 Memory Management Unit

A Memory Management Unit (MMU) relieves the operating system by implementing Virtual Memory Management (VMM), which is necessary for multitasking when the processes are not aware of each other and not explicitly designed to share the memory resources. Even in cases where the tasks are custom written to cooperate, virtual memory management can greatly simplify the memory model, and reduce software development time. For single-process systems, VMM can also improve performance or reduce memory requirements by allowing the software to run with just a subset of its total footprint loaded into main memory. This was actually the impetus for developing virtual memory, though it is rarely a necessity in modern embedded machines thanks to the decreased cost of memory. In fact, many embedded systems are designed without any secondary memory at all, just a primary memory large enough for the entire working set.

Virtual memory management in hardware requires, at a minimum, a buffer to store the page table, a few registers for control and configuration information, and logic to implement address translation. Such a minimalist approach, as found in the MIPS instruction set for example, requires that the operating system implement page flushes and the finer points of memory protection. More sophisticated MMUs may implement these themselves.

An MMU adds latency to memory accesses while table lookup and address translations are performed. Of course, any single-task embedded application is free of

this requirement; MMUs are only useful for multitasked environments where individual software programs are unaware of one another. In practice this limits their use to PDAs and other reprogrammable devices.

3.3.2.4 Direct Memory Access

Direct Memory Access (DMA) can exist within an embedded system at one or both of two distinct levels. Section 3.3.3.9 deals with DMA in processor internal memory. Usually, though, DMA is employed outside the processor level to place data from peripherals into main memory without having to involve the CPU. The processor is free to execute other instructions while the memory transaction is being performed. The major problem with this approach is that while the DMA transaction is occurring, the processor's cache cannot access the main memory. While the addition of logic within the DMA controller to prioritize the bus in favor of the cache may mitigate this concern, it cannot remove it entirely; the cache will still experience increased latency occasionally. Also, in the case that there is a peripheral which cannot be pre-empted (maybe because it receives data externally at a fixed rate), a processor may be forced to wait for the duration of the transaction. The net effect of DMA is reduced load for the processor, at the cost of some possible latency indeterminism.

In general, a DMA system reduces power consumed in the processor by relieving it of the burden of controlling the memory transactions. However, it may increase overall system cost and even overall power consumption, because it requires that each device on the memory bus have a DMA engine of its own. Therefore, only embedded systems with many memory transactions, such as networking (as discussed in Chapter 4) will benefit from the implementation of DMA. Because of the cache effect, it is inappropriate to include DMA in strict real-time systems.

3.3.3 Local Memory

Local memory refers to memory structures which are accessed at the latency of a single instruction, as compared to non-local memory which usually takes many clock cycles to

access. In most systems, local memory is on the same chip (die) as the processor itself, and non-local memory (covered in other parts of Section 3.3) is off-chip, usually in a separate package. For systems with a lower clock frequency, this is not necessarily so; as memory bus frequency decreases, so does the cost of access to off-chip memory. Caches are the most common and representative local memory structure, but the category includes stacks and scratchpad memories as well as some more exotic solutions.

In all computers, embedded or otherwise, the main purpose of any local memory arrangement is to supply data or instructions to the processor with as little delay as possible, ideally without forcing the processor to stall and wait at all. In embedded computers, there is the additional goal of supplying data and instructions to the processor at as little energy cost as possible.

3.3.3.1 DRAM

Traditionally, only SRAM has been used on-chip for local memories and caches, because of its manufacturing compatibility with digital logic. Although DRAM, with its analog sense amplifiers, was made available to the embedded market some time later, it was not widely accepted because of its much slower access times, despite its appreciable density advantage [54].

An inherent problem with DRAM is that memory cells that have been idle for some time must undergo a refresh cycle or lose their contents. This disadvantage alone is sufficient reason why embedded DRAM is still not competing with SRAM as a solution for primary local memory because of the added latency. However, adding an additional hierarchy level of on-chip DRAM, after SRAM, but before resorting to off-chip discrete memory can be an efficient technique [55],[56]. For one thing, embedded DRAM can be accessed with wider busses and at less energy than off-chip memory.

3.3.3.2 Stacks

The stack is an excellent example of a feature that was developed in the CISC era, excised from RISC architectures, and is currently making its way back into circulation.

Last In, First Out (LIFO) structures for parameter passing between subroutines, and for arithmetic operands were developed as early as 1959 [57]. A recent example is the Java Virtual Machine (JVM), a theoretical stack machine which must be implemented in software on any computer that runs Java bytecode without conversion [57]. Sun designed the JVM as a stack machine to simplify the compiler; as an object-oriented language, there are a large number of function calls and returns, and a stack is ideally suited for parameter passing and storing return addresses.

The use of stacks for high level language support and simple arithmetic operand manipulation is what led to their inclusion in so many CISC machines. They disappeared from RISC machines because of the difficulty of reaching non-top data, and the area cost of potentially unused memory, since the stack is seldom full. Stacks also create structural hazards for pipelining; an instruction which modifies the stack also modifies the location of the top of the stack, which is implicitly referenced by subsequent stack operations. Out-of-order execution is made extremely difficult because of the inability of instructions to reach their operands in arbitrary order; the result is an extreme limitation on achievable ILP.

Now we are seeing stack support increase in processors to facilitate fast context switching and simple parameter passing. At a minimum, support can be as simple as locating the stack in main memory and providing post- or pre-increment and -decrement addressing modes. ARM's Jazelle technology to accelerate Java processing places the top four elements of the stack, along with the stack pointer, in registers [58]. Stack bounds checking, explicit stack control instructions (such as summing the top two elements, or jumping to the address at the top of the stack, etc.) are more hardware intensive, but not uncommon. The highest level of stack support in a processor means keeping at least some of the stack locations on-chip for accesses at register speeds. However, if this hardware stack runs out of room and "spills" into main memory, access times for stack operations may not be determinate, which severely limits their effectiveness for time-sensitive tasks.

Because of their strength in parameter passing, stacks are likely to be beneficial in any embedded application with many function calls or especially interrupts. Real-time embedded control is a candidate, though care must be taken to ensure that stack faults such as spilling into main memory do not affect the latency of interrupt handling. The use of stacks for pure arithmetic reasons is undesirable because of the limitations imposed on ILP.

3.3.3.3 Cache Associativity

The simplest form of cache is a direct-mapped one, in which each cacheable memory line has only one possible placement within the cache. This arrangement leads to the simplest operation, and the least hardware overhead, but can occasionally lead to situations where a highly-used memory block is repeatedly swapped out for less-used ones. One solution is to allow alternative cache locations for each line, and to discard the least recently used (usually, though other replacement algorithms exist) block when adding a new one. Two-way and 4-way (or more) "set" associative caches imply that there are two or four cache locations *associated* with each set of memory locations. A fully associative cache allows any memory block to be placed anywhere in the cache.

Increasing cache associativity reduces cache misses and helps to ensure that the most valuable instructions and data are kept in the cache longer. It also requires overhead in terms of hardware and logic to determine the cache location of the addressed memory. The more associative the more "tag-matching" hardware is required. Additional overhead is required to implement the replacement policy that decides which block is flushed to make room for a missed block. Both of these overheads consume silicon area, and energy when used. As an example, we compare cache sizes and energy consumptions for 8 KB caches, with a block size of 16 bytes, implemented in a 0.25 μm process. The figures in Table 3.1 come from the CACTI cache exploration tool [59].

Associativity	Direct-Mapped	4-way	Fully
Access Time (ns)	1.451	1.714	4.044
Access Energy (nJ)	0.585	0.891	1.833
Total Area (cm^2)	0.022	0.024	0.047

Table 3.1: Cache Associativity Overhead

3.3.3.4 Cache Locking

In embedded systems, where the programmer is frequently given some degree of direct control over the hardware, there is an alternative to cache associativity for making sure that valuable code and data do not get replaced in the cache. The technique of cache locking allows the software to specify that a particular block be irreplaceable [60]. As long as that line is locked, it is unavailable for new data, which can limit the effectiveness of the cache if it is attempting to access other addresses served by that cache line. Higher associativity mitigates this effect by allowing the new data to be placed in an alternate location.

Cache line locking helps to address the non-deterministic access latency of caches for time-sensitive software. However, in order to do so, the programmer must be very careful that the processor stays inside the locked regions during the time-critical processing. Any accesses outside the locked regions have an increased likelihood of incurring time-consuming replace cycles.

Line locking is ideal for applications that might benefit from the flexibility of caching coupled ability to achieve time-determinism for critical periods. For example, a general-purpose processing system which must occasionally respond to real-time events might be found in data-recording systems. Avionics or industrial control applications are likely candidates.

3.3.3.5 Hardware Prefetching

It is possible to actively predict cache misses and preemptively fetch data or instructions into the cache, while idle, so that they are present when required by the processor. The

prediction techniques for cache prefetching are similar to branch prediction (discussed in Section 3.3.4.3). Just as with branch prediction, there are a number of potentially viable algorithms, the accuracy and effectiveness of which are heavily dependant on the application, and especially on the locality exhibited by the data or instructions. For example, "fall through" prefetching entails simply retrieving the next sequential cache line from memory when it is detected that the current one is nearly exhausted. [61] notes that fall through prefetching of instructions is not very successful when there are frequent subroutine calls, as with object-oriented languages, and that "target prefetching" is more useful. Target prefetching requires the addition of a hardware table to track predicted lines, similar to a branch prediction table. In general, instructions are easier to prefetch effectively than data due to higher locality.

Without prefetching, caches exhibit lower miss rates (though not necessarily less memory traffic) with larger line sizes; the larger lines exploit spatial locality, in a sense statically prefetching memory contents. When prefetching is employed, it is observed [61] that smaller line sizes (which reduce traffic) are more effective and result in lower overall miss rates. Smaller, less frequent memory transactions decrease the total power consumed by the memory system. As always, the magnitude of the effect depends on the application and data, and must be weighed against the cost of implementation.

3.3.3.6 Cache Miss Penalty Reductions

Associativity, victim caches, increased cache line size, and prefetching all fall under the category of optimizations intended to reduce the frequency of cache misses. The other, complementary, approach is to reduce the penalty incurred by cache misses.

In general, RISCs have lower miss penalties [25] than CISCs because there are generally more instructions in flight, and the processor can continue to do useful work while waiting for the memory to catch up. This effect does nothing to improve the speed of memory, but "hides" the latency incurred by the miss. Latency hiding is the general principle behind all miss penalty reduction techniques.

Some of these techniques are very straightforward. For example, the compiler (or in out-of-order machines, the issue stage) can hide potential data misses by scheduling instructions such that a potentially missed operand is not needed until several instructions later. If there is a cache miss, the memory system has had a moment to perform the fetch before the operand is needed, and some of the latency has been hidden. Such techniques are possible with instructions, too, but this is more difficult [62].

Another straightforward technique is to "wrap" the fetch of a cache line, so that the required address is fetched first and passed on to the processor. The memory controller continues to fetch the rest of the cache line while the processor is free to move on. An even simpler alternative is to simply fetch in order, as normal, but enable the processor to perform an "early restart" as soon as the required word has been fetched. The benefits of both of these techniques are greater when larger block sizes are employed in the cache [63]. These simple techniques cost very little to implement in silicon, and should be de rigueur in embedded processors for all applications.

A much more complex method of reducing the miss penalty is to add another hierarchy level, that is, another level of cache before resorting to main memory. The associativity, block size, replacement policy, writeback policy, memory technology, and physical arrangement of each level of cache (potentially, twice each, with Harvard units) need not be the same. The total design space of a multi-level cache system is very large indeed, and provides much opportunity for optimization.

While adding cache is almost always beneficial from a pure performance perspective, it is less straightforward to determine the correct mix of L1 and L2 cache. Short and Levy simulated a variety of two-level caches [64] and while they did not directly make the conclusion themselves, it is evident from their data that in almost all cases it is a more efficient use of chip real estate to extend the L1 cache rather than using the same area to create an L2 cache. However, access times for cache increase with capacity and impose an upper limit to the size of the L1 cache if it is to be accessed with single-cycle latency. Beyond that size, it is far more desirable to add L2 cache if manufacturing budget and die area permit, rather than accepting increased latency in the L1 cache. Jouppi and Wilton

supported this and expanded on the theme in [65], asserting that the addition of a second level cache is more beneficial when the memory cells in the L1 cache are multiported and therefore larger than the cells of the L2 cache. Hundal [66] used a theoretical, formulaic approach to model a system with SRAM L1 cache and DRAM L2 cache, and shows how to calculate the optimal mix.

These major modifications to the cache hierarchy, as well as careful selection and tuning of the properties of each cache level, are complex and have been studied at length. The ramifications for specific embedded architectures and workloads are not yet completely understood, however. For this reason, we recommend that embedded designers pay careful attention to the matter, and that future research on the topic be conducted by the academic and commercial communities.

3.3.3.7 Scratchpads

Directly addressable memories, known as scratchpads or Tightly Coupled Memories (TCMs) at the first hierarchy level can have several advantages over cache. Most significantly, such memories are always more area- and power- efficient than cache for any given storage capacity, because they lack the hardware required by caches for tag matching, address decoding, and replacement. They suffer one major disadvantage, however; scratchpads require direct software or compiler control over their contents, whereas a cache (excepting cache locking and software prefetching) is entirely autonomous. Scratchpads suffer from a relative lack of research regarding their optimal use, compared to the much better-understood cache model, because they are limited to applications for which hardware and software can be co-developed.

In such systems, including most embedded applications, the potential for performance increase is worth the non-recurring engineering cost of analysis and development. This sort of analysis, and the tradeoff between caches and scratchpads is investigated in great detail in Chapter 5.

3.3.3.8 Loop Caches

A very small cache (as small as only 64 entries) that exists either beside the main cache, or between the main cache and the processor is called a *filter cache* **[67]** and is cheaper and faster to access than the main cache. The small, directly mapped cache constantly reloads as a program passes through it, and is accessed instead of the main cache in the event of a very small backward branch or loop. Even if the entire loop exists and remains in main cache for the duration of the loop, it is still more efficient to access the smaller loop cache. The immediate and obvious shortcoming of a filter cache is the constant overwriting of its contents, even for code sections which are not executed in tight loops.

A cache of this nature that recognizes and selectively loads only loops is called a *loop cache* and can be loaded either dynamically [68] or statically with the help of the compiler [69]. Either way, the bounds of the current loop are generally stored in registers, which further reduces the access cost by rendering the tag matching hardware irrelevant. The major drawback is that they are not energy efficient; the cache must continuously determine whether the current code is a loop, and consume energy in doing so. For this reason, they are unsuitable for handheld or battery-powered embedded systems.

3.3.3.9 DMA Overlay

DMA can be an alternative to cache for getting data and instructions into the local memory. Such a system is called a DMA "overlay" and can have significant benefits over a cache-based system. The first such benefit is actually a removal of one of the primary limitations of cache: the indeterminism introduced when a cache misses and stalls the processor during a replacement. DMA overlay requires that the programmer be involved with timing memory transactions, but with that added programming effort comes greater control. Second, a DMA system could be used to perform buffer manipulations such as interleaving, rotations, etc. without tasking the processor. In some multimedia applications, this can be a significant relief to the processing budget [62]. The tradeoff to be considered is between, on one hand, increased programmer effort and

hardware complexity to implement a DMA overlay, and on the other hand, reduced efficiency and indeterminable access latencies for cache.

3.3.4 Exploiting Instruction-Level Parallelism

Instruction Level Parallelism (ILP) is the concept that describes the mutual independence of some sequential instructions in a program. Exploiting such parallelism by simultaneously executing instructions when possible is a powerful tool in modern computing. Several techniques can be used, which we will explore next. It should be noted here, though, that these techniques in hardware have a negative effect on the ability of the programmer to schedule tasks with absolute precision. As Edward Lee writes, "many recent computing advances do more harm than good when embedded computing systems absolutely must meet tight timing constraints." [70] In that article, Lee refers specifically to speculative execution and dynamic dispatch (out-of-order execution) which are both discussed in this section. In general, the more aggressively a processor employs the following three techniques, the more indeterminacy is added, and the less suitable the processor is for nano- and micro-second scale time precise computing; for example, as Lee argues, network processing.

3.3.4.1 Pipelining

Increasing the pipeline length (number of stages) in a processor is usually done with the intention of increasing the clock frequency. If properly balanced, the amount of work done by each stage of the pipeline is decreased, and the corresponding clock period is reduced as well. The clock frequency is increased, and the overall throughput of instructions through the processor per unit of time is increased.

However, there is overhead involved in each pipeline stage, as the instruction and its operands (in whatever intermediate form they exist in at each particular stage) must be moved along the pipeline registers before the necessary operations can be carried out. Additionally, the increased number of instructions in flight increases the probability of dependencies between instructions. These dependencies not only limit the actual

throughput of instructions by introducing stalls into the pipeline, but they also necessitate more logic dedicated to detecting and handling the dependencies. The additional logic must operate at the speed of the pipeline, or else it reduces the clock frequency. Regardless, that additional logic consumes energy and incurs manufacturing cost.

A long pipeline can complicate other specific architectural goals as well; the penalty for a mispredicted branch is directly related to the length of the pipeline, and can reduce the accuracy of the cache as well as the reactivity of the system. Between this property, and the higher cost and power consumption incurred, aggressive pipelining is not suitable for applications with a high mix of branches and control instructions, such as control systems. It is more valuable in applications with long, predictable streams of instructions, and long loops. For example, 2D video applications can be ideally suited.

Pipelining is commonly employed in special functional units as well, especially floating point arithmetic units. This pipelining is not as susceptible to control hazards, since each functional unit (and each pipeline) executes only one type of instruction. However, branches and data dependencies may still occur in the processor external to the functional unit and a long pipeline increases the impact of those events on overall performance.

3.3.4.2 Superscalarity and Out of Order Execution

The ability to issue multiple instructions simultaneously greatly increases the complexity of the system, but can exploit ILP to a greater degree than simple pipelining. Dynamically tracking data, control, and hardware dependencies requires sophisticated algorithms and significant logic resources, and grows exponentially more difficult as the number of simultaneous issues increases. Since the amount of ILP available is a characteristic of the software itself, even the best dynamic multiple issue schemes rarely achieve perfect efficiency, and the returns tend to diminish as the degree of scalarity increases.

Out-Of-Order (OOO) execution allows instructions to be issued without waiting for the previous instruction, as long as the functional units are free to service it. In pipelined

machines, this increases the exploited ILP by reducing the impact when an instruction in stalled pending resolution of a hardware or data dependency. Usually, the same algorithm that schedules multiple instructions for simultaneous issue is capable of issuing instructions out of order. The addition of OOO capability to any superscalar machine greatly increases ILP, particularly if instructions can be executed speculatively, which is not a great deal of extra complexity if already tackling the much more difficult problems of superscalarity and OOO. For this reason, most dynamically superscalar machines (we have excluded VLIW and DSPs which are often statically superscalar) are also OOO and speculative.

Without very careful tuning of the pipeline and multiple/OOO issue logic, the complexity of the task can lead to very long critical logic paths and greatly limited clock frequencies, all for a less-than-optimal return on Instructions-Per-Cycle (IPC). Especially when considering the extra chip area and energy consumption, superscalar and OOO execution are usually less efficient than simply increasing the clock frequency. This is especially true in embedded systems design, since the code can be fine-tuned for performance to a degree not possible in general-purpose processing; much of that fine-tuning is defeated by dynamic multiple issuing.

According to LSI Logic, "out-of-order execution is often undesirable in timing critical, DSP related applications." That quote comes from an article [71] about a DSP processor, but it is true of any timing critical application, especially control. OOO and superscalar are best suited for devices without heavy power or cooling constraints, and very large processing requirements. This is not typically the case in embedded computing, which is why the two technologies are rarely found in embedded devices.

3.3.4.3 Branch Prediction and Speculative Execution

In [72], David Wall identifies branch prediction as critical for any processor intending to exploit ILP. If branches occur as frequently as every fifth or sixth instruction, as with many non-numeric programs, it is difficult to keep a heavily pipelined or superscalar processor busy. Put another way, the extractable ILP is limited to the lesser of the

number of simultaneous instructions in hardware, and the number of instructions in a basic block. Branch prediction (and speculative execution) is a way of concatenating basic blocks to form a longer string, a "predicted block," of uninterrupted instructions, provided the prediction is correct. The length of the predicted block is dependant upon the accuracy of the prediction scheme, according to Equation 3.1 [73].

$$size \propto \frac{1}{1 - accuracy} \tag{3.1}$$

There are a wide variety of branch prediction algorithms and hardware schemes, including static solutions performed by the compiler. We do not detail all possible algorithms here; our intention is to introduce the different options available to embedded designers, and comment on their applicability.

- **Fixed Prediction** schemes simply assume that all branches are taken, or not taken, and have a fairly low accuracy rate. Such schemes are usually only a slight improvement over simply waiting for the branch to be resolved. There is one case that can be predicted with high accuracy, however: almost all short backwards branches are taken (usually, these branches terminate short loops).
- **Simple Counting** schemes track the past behavior of a branch by associating a very small (one or two bits) counter with the address of the branch. The branch is predicted to have the same effect as the last time it was encountered. This method yields a great improvement in accuracy over fixed prediction schemes, but requires a table of branch addresses, counters, and logic to look up the branch and its prediction from the table.
- **Correlated** prediction schemes track not only the most recent behavior of the branch, but its pattern in recent history. This is achieved by the addition of another factor, such as the decision pattern of other recent branches, in addition to the address of the current branch [74]. The number of registers and amount of logic required to implement correlated prediction is higher, but for some software, the accuracy can be as high as 97% [75].

The logic required for branch prediction can add considerable latency to that stage of the pipeline, which can potentially limit the maximum clock frequency or necessitate a stall. The addition, in any prediction scheme, of a Branch Target Buffer (BTB) that contains the predicted program counter after branch (whether taken or not) can relieve some of the latency, but doubles the size of the prediction table. Even more latency can be avoided if the table also holds a copy of the instruction located at the predicted program address. Again, this increases the size of the table.

A complementary technique for reducing branch penalties is more common in DSPs. Branch delay slots are placeholders in an instruction stream, immediately following a conditional branch instruction. Into these slots are placed a number of instructions that will be carried out after the branch, whether it is taken or not; they are immune to the effect of the branch. When possible, the compiler fills the delay slots with instructions whose results do not affect the condition of the branch. Another way to view this technique is to imagine that the branch instruction is fetched and initiated "early" by the processor. While the condition is being evaluated, the processor need not be stalled; it can execute the instructions in the delay slots, without ramifications on the branch. Ideally, the branch will be resolved while the delay instructions are being processed, and the front end can fetch the correct instruction following the branch, without any noticeable delay in the pipeline. From a hardware perspective, this is simple to accomplish, but its effectiveness is limited by the number of suitable instructions available compared to the branch penalty (which increases with pipeline depth and scalarity).

Similar to delay slots is the technique of speculative execution, which reduces the branch penalty in pipelined machines by guessing at the result of the branch and continuing to fetch and execute instructions along the assumed path while the branch is still unresolved. The speculative instructions are processed, but their results are not committed until the guess at the branch has been verified; if the guess was incorrect, the results of the speculative instructions are discarded. Speculative execution is effective if the prediction accuracy rates are high enough, but exacerbates the penalties for a

misprediction. The impact on hardware complexity is significant: storage is required for intermediate results before the speculative instructions are committed, as is complicated logic for operand and results multiplexing. However, a processor already capable of out-of-order execution can be extended for speculative execution quite simply.

Lastly for this category, conditional or "predicated" instructions whose results are only accepted if the condition is satisfied are used to remove branches completely in some cases. Tyson [76] reports between 5% and 30% of branches are removed from SPEC benchmarks, depending on the aggressiveness of the compiler. This is an important result because it increases the length of the basic block, and therefore the available ILP to be extracted through other means. Tyson reports an increase in average basic block size by as much as 40%. It should be noted that the actual work done to effect these change in the code is done by the compiler, not the processor, though the processor must be capable of handling predicated instructions.

In embedded systems design, where it is reasonable to fine-tune software to a degree unrealistic in general-purpose computing, all static methods available should be employed as aggressively as possible, as long as support for the methods is available in hardware. Dynamic solutions are a matter for further analysis, however, since their implementation affects runtime performance of the system. Dynamic branch prediction and speculative schemes are more likely to be efficient in systems employing significant ILP-exploiting hardware: long pipelines and superscalar execution. Since such hardware is not advised for control or real-time applications, they are unlikely to benefit from branch prediction or speculative hardware.

3.3.5 Vector Processing

Single Instruction Stream, Multiple Data Stream (SIMD), or Vector, machines have multiple parallel instances of the functional units in the ALU which operate simultaneously on multiple operands, when only a single instruction requests that they do so. Code that repeats the same computations, operations, and comparisons over a number

of data points (typically, *for* loops where the counter is an index to an array of data) can be processed in parallel and the number of iterations and instructions decreased. The concept, Data Level Parallelism (DLP) is similar to ILP, but seeks instead to find repetitions at the data level. Unlike ILP, DLP is usually detectable by the compiler, which can perform code transformations such as loop unrolling in order to exploit it. Even in I/O based systems, which operate on unknown data streams, such as cellular phones or automotive controllers, the DLP is evident from the code at compile time as long as the execution path is not heavily dependant upon the input data.

Vectorization has the effect of reducing the instruction count of a suitable program, reducing memory footprint, but more importantly reducing instruction memory bandwidth. Lee and DeVries report a full order of magnitude reduction in the number of instructions executed (and therefore fetched, albeit mostly from cache) [77]. One study on the performance of Intel's MMX instruction extension (which is not the same as a full vector processor) reported instruction count ratios between 0.62 and 9.92 for MMX versus standard compilation [78]. Most of the applications tested showed significant improvements; the two applications (g.722 voice compression and jpeg image compression) which showed negative results were attributed to excessive function calls and the nature of the MMX libraries as opposed to fully compiling MMX code. In general for vector units, higher data bandwidth is required in order to realize faster execution, but overall the total memory traffic is reduced significantly.

The extra functional units and wider datapaths come at a cost to die area, but [77] reports that vector processors are smaller than similarly performing superscalar machines. Broader datapaths complicate clocking networks on the chip as well, resulting in a lower maximum clock frequency than might otherwise be possible.

In 1999, Motorola added Altivec to their high end PowerPC chips; Altivec comprised 162 new instructions, a 32-entry 128-bit vector register file (divisible into 8- 16- or 32-bit values), and a vector execution unit. In order to provide data bandwidth to this wide execution path, they also found it necessary to add a new memory controller [79]. The vector extension to SPARC chips is named Visual Instruction Set (VIS). Intel's is

MultiMedia eXtension (MMX). AMD's is named 3Dnow!. The names clearly highlight the multimedia (and in particular, 3D graphics) workloads at which they are targeted. Embedded systems with similar workloads are likely to benefit greatly from such improvements, while systems with very linear (low DLP) data streams, or little data processing at all, will not benefit. There has even been some interest, recently, in using special purpose Graphics Processing Units (GPUs), which have traditionally been coprocessors in desktop systems, as the primary processor in specialized applications [80], [81].

3.3.6 Multithreading

Multithreading is the ability of a processor to process more than one distinct flow of instructions with little or no overhead. This can be achieved by reducing the overhead involved in alternating between threads, or processing them concurrently, as Simultaneous Multithreading (SMT). Each thread has a "context" associated with it, which includes the current data, the current state of the processor, and the program counter. An SMT processor usually contains multiple contexts at a given time, and each instruction updates only the context that corresponds to its thread [82]. Consecutive multithreading usually concerns itself with optimizing the process of switching contexts.

Deep pipelines complicate thread switching by increasing the number of elements which must be reconciled with a context switch. Memory systems are also affected: useful objects in the cache for one thread are likely useless for another, and rapidly changing threads can ruin the cache's efficiency for both. With SMT, resource scheduling must be carefully implemented. In between instructions in one thread, another thread may alter the contents of memory or the register file, or some peripheral device. Mutual exclusion and reliable passing of semaphores are vital in these systems. Due to the complexity of thread synchronization, and as a result, the indeterminable state of shared resources, multithreaded systems are not well-suited to real-time systems [70], such as industrial controllers or avionic and automotive computers. The ability to handle multiple independent tasks is more useful in user-oriented devices, such as cellular

phones or PDAs, or in instrumentation monitoring (provided the time constraints are soft ones). Multiple threads can also be useful in multimedia applications. For example, MPEG decoding can be broken into several distinct processes [39].

An interesting side effect of multithreading is the low-level shift in the primary function of the cache; in single-thread processors, caches exist to reduce latency in memory accesses. In a multithreading system, the latency incurred by a cache miss (or any other long stall) is hidden by switching threads. The cache's primary function in such a system becomes one of keeping the maximum number of threads in a ready state at any given time, which simplifies to maximizing bandwidth rather than reducing latency. Prefetching, similar to what is described in Section 3.3.3.5, can be a great benefit in this case [61].

3.3.6.1 Register Windowing

An alternative to stacks for fast context switching between processes is the technique of register windowing, whereby a processor has a very large register file, but only allows access to a subset (window) at any given time. The windows overlap, which makes parameter passing very simple but reduces the number of protected registers for each context. Register windows are a defining feature of the SPARC architectures, and have been criticized there for slowing down performance of the processor by introducing latency into every register access in order to decode the register address and multiplex them all to the functional units.

3.3.7 Special Instructions

Custom FIFOs, barrel shifters, Content Addressable Memories (CAMs), and many other types of specific hardware constructs may be added to a processor, and accessed by means of additional opcodes added to the instruction set. There are many examples of these custom instructions. Some of the Motorola 68K processors had instructions to handle Binary Coded Decimal data which is very useful for, for example, Cobol processing. Since binary functional units are more efficient, more recent processors are

forced to translate decimal data types into binary, perform arithmetic, then convert the binary solution back, requiring considerably more instructions.

In the Intel x86 ISA, there is an instruction named REP SCAS that searches for a byte pattern in a memory array, returning the address [83]. This instruction is irreplaceable for text processing, or any other unsorted search, but is unwieldy to support in hardware. MIPS, otherwise a posterchild for the "reduced" instruction set camp, includes a reciprocal square root instruction, which is all the more remarkable for the fact that it gives up a strict interpretation of the IEEE 754 floating point standard in order to complete more quickly. SPARC has one instruction that directly affects memory and is not a simple load or store: the CASA instruction compares a register value with the value at a memory location, and if they are equal, swaps the memory location with a second register. This extension of an atomic test-and-set instruction greatly simplifies synchronization and resource sharing algorithms.

The potential variety of such hardware additions is endless, limited only by the imagination of the hardware designer, and the usefulness of the instruction in the final application. The more exotic the instruction, the more specialized the processor becomes to the target application. Turley explains how the TBLS instruction for looking up (or making up) data points on an arbitrarily specified two-dimensional curve is probably a useless feature for most applications, but irreplaceable when doing motion control [84].

An approach by soft processor core developers Tensilica [85] and ARC [48] is noteworthy here. Both companies' configurable processor cores allow designers to add hardware functionality and instructions to the ISA. The custom instructions are completely specified by the developer, so that the potential variety is unlimited. The new instructions are supported by the software development tools, including the compiler, to allow full benefit for the entire product design cycle. Such completely customizable platforms are ideal for the optimizations and development espoused in this thesis.

3.3.7.1 Floating and Fixed Point Arithmetic

The most notable example of specialized functional units (with special instructions) is that of floating point arithmetic. The majority of the most popular processors either come with floating point capabilities built in, or provide it in an optional configuration. For example, the ARM9TDMI core does not have floating-point capability; many but not all of its derivative processors, however, do have the capability to perform floating point operations. Many sophisticated processors, especially superscalar ones, have separate pipelines for dispatching floating point instructions, because of their longer execution time. Floating point is essential for 3D graphics rendering, physical simulation, and for motion estimation and control.

Fixed point arithmetic has many of the same considerations, but is much less common in non-DSP processors. It is used primarily when fractional numbers are required, but limited precision is acceptable in exchange for faster computation.

3.3.8 Interrupts

Most processors have the facility to respond to external and/or internal events, or interrupts, and break the normal flow of software execution in order to respond appropriately. For some embedded applications, especially control (as in consumer or industrial controllers) and data gathering, interrupts are a vitally important feature. Most external sensors and user input peripherals are integrated with the processor in this manner, including those whose data is transferred via DMA (in which case it is the DMA controller that directly triggers the interrupt). For systems with heavy real-time requirements, it is important that multiple interrupt sources be maskable or prioritized, so that the system can effectively meet critical deadlines without breaks.

Internal interrupts or traps are usually triggered by error conditions within the software. An attempt to divide by zero, for example, or access a prohibited memory location, can cause the processor to trigger an interrupt, which allows the programmer to specify error-recovery behavior. This is an essential mechanism for ensuring that software does not cause spurious effects and bring down the system, especially when

dealing with unpredictable data sets or multiple inputs whose relationships are difficult to specify.

Another type of internal interrupt can be triggered by counters which fire after a user-specified number of events. Counters whose events are necessarily time-linear are a special case, called timers, which are essential in most real-time systems, or for task-sharing operating systems. Finally, a special type of timer called a "watchdog timer" is often included in embedded systems as a last resort for system stability; the watchdog must be explicitly and regularly reset, or else it fires and causes the system to reboot. In this manner, watchdogs prevent systems from remaining in infinite loops or other locked conditions for excessive lengths of time.

3.3.8.1 Precise Interrupts

A processor handles interrupts *precisely* if the state of the processor at the time of the interrupt is entirely consistent with the last instruction (from the normal or non-interrupted instruction stream) issued. The trivial case, a non-pipelined, in-order processor which completes each instruction before issuing a new one, is incapable of imprecise interrupts. On the other hand, a pipelined or superscalar processor which has multiple instructions in flight at any given time may have difficulty achieving the *precision* in interrupt handling. If an interrupt is imprecise, the burden of ensuring the correctness of the processor state becomes an additional burden for the interrupt handler; this extra work can limit the reactivity of the system.

As an example, consider a timer-driven task scheduling system. When a time slice expires, the timer interrupts the currently executing task and the interrupt handler passes control to the OS, which determines the next task to execute. As time slices decrease in size, the proportion of processing overhead consumed by the interrupt handler increases. If the interrupts are imprecise, the additional overhead required to ensure that the running task is 'parked' in a correct and predictable manner can become a significant system cost.

There are two major approaches to implementing precise interrupts [86]; the pipeline can simply be flushed, either allowing the instructions to complete and update the

processor state, or discarding the in flight instructions. Alternatively, extra logic and state registers can be included for more complex bookkeeping in the event of an interrupt. The silicon overhead of the first approach is minimal, but it increases latency and decreases reactivity of the processor to interrupts. The latter approach incurs more silicon overhead, but allows interrupts to be handled immediately, which can be critical for some real-time systems, and the reactiveness required by many embedded systems.

3.4 Chapter Conclusions

The preceding sections demonstrate the breadth of the design space for embedded hardware. We have endeavored to provide real world applications as practical examples of each feature or optimization. However, it is impossible to define a complete list, either of suitable applications for each feature, or of suitable optimizations for each workload, for two reasons. The first is the sheer size of the design space; there are simply too many combinations to list. The second is the diversity of end product applications, and the (sometimes conflicting) requirements that result; complex systems require more detailed analysis and profiling in order to select or design the optimal hardware.

In the next two chapters we address these complications with case studies. Chapter 4 examines networking and inter-computer communication as an embedded workload, and goes into greater depth in exploring the application of the features in this chapter to that end. Chapter 5 takes the complimentary approach of focusing on a single optimization. The specific tradeoff of cache versus scratchpad memory is investigated, and a method of profiling applications in order to find the optimal tradeoff point is proposed.

4. Architectural Support for Networking

4.1 Chapter Introduction

There are two phenomena at work in the ongoing evolution of computer communications. The first is that an increasing number of devices, embedded in an increasing range of applications, are communicating. The second is that data bandwidth requirements are increasing exponentially.

4.1.1 Low Bandwidth Devices

Component stereo systems send music digitally and wirelessly to one another. Videoconferencing and telecommuting are commonplace and invaluable to many offices. As we enlist computers into more and more roles [87], we increasingly demand that data move seamlessly between any individual devices that need it.

Embedded computers exist in environments where bandwidth is limited either by processing power, power budget, or environmental constraints such as noise. The dominant example is cellular telephony, where maximum data rates under ideal conditions are in the neighborhood of 400 kbps (for EDGE; real-world typical performance is closer to 150 kbps [88].) In such cases, it is primarily economics that drive the optimization of networking in the processor. Optimization of networking decreases the required processing performance and energy consumption of the total system and can reduce the number of communication-specific components (along with bussing between them). This frequently leaves the processor responsible for lower communications layers, up to and including the data link layer (OSI level 2), which would otherwise be handled by dedicated network interface hardware. If these tasks make up a significant portion of the processor's total workload, it makes sense that the processor be selected or optimized to reflect this.

4.1.2 Increasing Bandwidth

Nearly everyone is familiar with the oft-cited and ill-named Moore's Law: processing power doubles every eighteen months. There is a similar "law" describing the increase of communication; Gilder's Law predicts that data bandwidths *triple* every *twelve* months [105]. When Moore's law is invoked, it is usually to laud the stunning rate of advance in the semiconductor field. Here, however, we expose it as a limitation. Already, a modern processor connected to Gigabit Ethernet (GigE) uses up to 60% of its processing power just for protocol handling [90]. In wired applications, such as gaming consoles or surveillance systems, bandwidth is not limited by the form or location of the computer. In these systems the networking bottleneck is not the transmission medium; it is the processor [91].

SCADA systems increasingly utilize Ethernet as a transmission medium [92]. The bandwidth demands of such systems vary widely, and in many cases are low enough to be included in the category of Section 4.1.1. In higher bandwidth cases, and even in some of these low bandwidth cases, however, the synchronization and latency requirements are strict enough to demand architectures similar to high bandwidth machines. In light of such increasing demands on latency and bandwidth, and faced with Moore's law and its implications of physical limitations on computing power, we come to the conclusion that simply increasing clock speed of the CPU is not sufficient. In order to keep up with networking, we must employ architectural optimizations and dedicated hardware features to relieve some of the burden.

4.1.3 Chapter Outline

This chapter investigates, using the background developed in Chapters 2 and 3, the architectural issues involved with computer communication and networking. Where applicable, both the high and limited bandwidth cases will be discussed. We begin in Section 4.2 by revisiting the topics of Chapter 3 in specific relation to communications

and networking in embedded systems. In Section 4.3, we present specific commercially available processors and comment on their suitability for networking applications.

4.1.4 Network Processors

This chapter is not concerned with Network Processors (NPUs) or Communications Processors (e.g., Freescale PowerQuicc) whose sole function is the handling of high speed communications traffic. Such processors are usually highly optimized SIMD devices, with diminished programmability, and are not generally suitable for embedded applications other than specifically as network traffic movers [93]. In other words, while they are "embedded," NPUs are themselves the whole application, as opposed to being a part of a larger device. Their design is sufficiently different from the discussion in the rest of this thesis as to lie outside its scope. The rest of this chapter will omit this class of processor in favor of processors and applications for which communications and networking is a significant (even primary) but not definitive portion of the total workload.

4.2 Architectural Issues

In this section we discuss the architectural optimizations, features, and decisions described in Chapter 3, as they pertain to networking and computer communications in embedded systems. The organization is based loosely on that of Chapter 3, but omits many topics that are not relevant or do not have a direct effect on networking.

4.2.1 Generalizations and Broad Issues

We have already addressed the issue of power consumption in the introduction; low power devices, usually handheld, tend to have modest bandwidth requirements and can tolerate latency. Networking optimizations for such devices should emphasize power and processing efficiency. The devices with higher bandwidth requirements (greater than

megabit per second) are limited by processing power; optimization for such devices must focus on minimizing latency and maximizing throughput.

Along those lines, a broad observation can be made about Harvard and von Neumann architectures. Harvard machines are capable of higher throughput than von Neumann machines, which tend to consume less energy. It is reasonable to conclude therefore, that in the general case, Harvard architecture is better suited to high bandwidth applications than von Neumann, and vice versa for the lower power, low bandwidth case.

The RISC versus CISC tradeoff is a recurring theme in the following sections. Insofar as RISC correlates to higher clock frequencies, however, it can be said of the general case that CISC machines are more appropriate for low bandwidth, low power applications. This will be reinforced in Section 4.2.6, when we discuss the advantage of specialized instructions for communications workloads. However, the RISC/CISC debate is alive and well for pure performance, and it is difficult to draw a direct conclusion for high bandwidth applications.

Regarding clock frequency and power consumption, we note that in systems with high bandwidth requirements (and low latency tolerance), voltage or clock scaling to reduce power consumption is undesirable [70]. It takes too long for the system to recover from low-power modes and resume full speed operation. However, low power systems that are most likely to employ these advances usually have much lower bandwidth and latency requirements, and can tolerate the slow recovery.

4.2.2 Memory

The processor (sometimes via an operating system) is responsible for the transport-layer protocol work, which implies calculation of checksum to verify the packet, and ensuring correct packet sequencing, amongst other things. These transport-layer duties must be performed at a rate determined by the network, lest the Media Access Control (MAC) layer buffer overflow. Applications, however, require data at their own rates, though, and so the payloads must be rebuffered until the application is ready to use them. A buffer of some optimal size is required for the MAC protocol, another for the transport protocol,

and one for each transport-client application, each of different sizes and with different throughput rates.

Adequate buffer space reduces the burden on the processor by allowing multiple packets to be handled at once, and by eliminating the necessity of constantly monitoring the network interface hardware, if it exists in the system. (In systems where the processor handles these duties, adequate buffering is even more essential.) Additional buffering is required at the application layer to allow the application (or applications) to process data at its own optimal rate.

4.2.2.1 Memory Hierarchy

Hennessy and Patterson [25] note that when interfacing with the network interface hardware, it is wise to minimize the number operations with slower, off-chip memory. Therefore, status information which will be monitored most frequently should be kept in locations which are accessed quickly. Mapping network interface status information to registers is feasible, particularly if the interface is polled rather than interrupt-driven; a polled interface is checked more often than an interrupt-driven one.

The next level of hierarchy, cache, is generally too small to keep entire packets in high bandwidth communications protocols (though it may be acceptable in low bandwidth cases). If only modest buffering is required, because inter-arrival delays are high, or packets are small enough, embedded DRAM is a good solution. Greater buffering requirements will necessitate the allocation of off-chip memories such as SDRAM for packet payloads. However, in such cases, it may still be desirable to locate the headers in cache. In any case, care should be taken to match the block size of the cache to the granularity of the cached objects, in order to minimize wasted cache replacement activity [91]. Block size, along with the other cache parameters such as replacement policy should be evaluated on a case-by-case basis for each application.

In interrupt-based systems, cache line locking or better yet, scratchpad memory may be useful to hold the interrupt handler code, to reduce latency and avoid wasted cycles by

preventing the processor from having to refetch every time an interrupt occurs. As discussed in Chapter 3, line locking is most useful with caches of higher associativity.

4.2.2.2 Memory Bus

The discussion in this section applies primarily to high bandwidth systems. In lower bandwidth systems, the memory bus is not as significant a bottleneck.

Network data is passed back and forth between memory and the processor multiple times. Employing Direct Memory Access (DMA) to move data between the Network Interface Controller (NIC) and memory directly is an improvement over any model that requires the processor to actively transfer the data. Using the processor to arbitrate these transactions is inefficient because the entire processor is prevented from doing other useful work, even though most of the functional units of the processor are not employed in the data movement, and because the processor is forced to operate at the speed of the I/O bus, rather than its native clock speed.

An alternative approach endorsed by Mukherjee and Hill [94],[95] requires computer architects to rethink the way the network is conceptualised by the computer. Rather than treating the network interface as a hard disk, which requires OS traps and address virtualization, the network interface hardware could reside directly on the memory bus and be immediately addressable by user processes. The memory bus is between two and ten times faster, in terms of both data rates and latency than the I/O bus [95], where the NIC usually resides. The concept is called User Level Network Interface (ULNI), and allows for efficient caching of both network data and network interface registers. Intel, Compaq, and Microsoft have developed a ULNI specification called Virtual Interface Architecture **[96]**.

4.2.3 ILP

The techniques for exploiting ILP all come at a cost of increased complexity and therefore, usually, decreased power efficiency. As a result, the more aggressive the technique, the less likely it is to be suitable for low power devices of the sort that have

modest communications requirements. Instead, such techniques are usually found in architectures aimed at high performance. However, as we have pointed out in previous sections, the higher the communication requirements, the tighter the real-time constraints on dealing with network traffic. Deep pipelining, dynamic superscalar and/or OOO issue, and branch prediction all do more harm than good when a system must meet tight timing constraints [70]. CISC architectures, which place more emphasis on work done per instruction, rather than more instructions (and ILP) per unit of time, are recommended.

4.2.4 Vector Processing

In normal single-stream communications,[1] there is not much data level parallelism to be extracted in the processing of communications protocols, aside from a few specialized operations, such as the calculation of Cyclic Redundancy Checks (CRCs) or checksums, or block memory transfers. These will be discussed in Section 4.2.6. As a result, vector processing is unlikely to be of any advantage for communications purposes.

4.2.5 Multithreading

Simple embedded devices which lack dedicated network protocol hardware rely on the main processor to process all OSI communication layers. However, since there is usually only one client (application level) task, a single thread can be used to recover the payload from the entire protocol stack, especially if the communications protocols employed are relatively simple, (e.g., Controller Area Network (CAN)). There is some evidence to suggest, in low bandwidth cases where some latency can be tolerated by the application or absorbed by sufficient buffering, that individual threads to receive, process, and

---

[1] It should be noted that the opposite is true for communications processors and network processors that may carry out identical operations on several packets or communications streams simultaneously. However, as noted in Section 4.1.4, this thesis is not concerned with such architectures.

transmit packets (in addition to the threads responsible for the end application software, if any) could lead to more efficient utilization of processor resources [97].

When real-time or low-latency response to network traffic is required, the interaction between independent threads is too difficult to predict and is a barrier to efficient communication [70]. An alternative exists in some recent embedded programming environments, (e.g., MathWorks Simulink), which provide concurrency without processes or threads [98].

4.2.6 Special Instructions

In this section, we explore specific instructions and operations which exist on some commercially available processors which are of great benefit to networking. Of course, when designing custom hardware, it is sometimes possible to include circuitry to perform some of these operations specifically. One could even add an entire protocol stack; it has been demonstrated that an entire Internet Protocol stack can be implemented in around 10K ASIC gates [99]. Addition of all, or a subset of the required hardware to a conventional 32-bit embedded processor, whose sizes range from 30K to 600K gates [100], seems to be economically feasible.

4.2.6.1 Block Memory Instructions

RISC architectures typically move data in small units: one instruction to load the data into the processor, another to store it at some external memory location. To move more than a word of data, each instruction must be called once per data word. CISC architectures, on the other hand, frequently have block-move type instructions which expedite the transfer of large amounts of data.

Because of its limited data-movement capabilities, a typical RISC processor spends the majority of its cycles transferring packet payload between memory and the network interface [101]. A CISC machine is more effective at moving data, and therefore has been found to be more efficient when packet sizes are large, and payload transfer dominates [102]. RISC machines, with their higher throughput of simple arithmetic

instructions perform the header analysis better on small packet sizes. Better still would be a blend of the two: a RISC-style processor with the CISC-like addition of a block move unit, such as the one available in the VIS extension to Sun's UltraSPARC [91].

4.2.6.2 Switch Statements

Most of the fields in a packet represent a small number of discrete choices and are best handled by switch statements, which compile to a very long series of comparisons and branch statements. It is essential, therefore, that a processor doing protocol analysis have a wide variety of conditional instructions. In addition, the long decision trees one is likely to encounter make branch prediction very difficult, and degrade the performance of otherwise efficient processor pipelines.

Some processors have recognized the abundance of switch statements in networking; the second generation of ARM's Thumb architecture extension is intended specifically to optimize networking in embedded applications. Thumb-2 contains a "table branch" instruction for switch statements that generates branch targets directly from a table of program-counter relative destinations from offsets in the register file. Properly used, this instruction can reduce compilation of a switch structure in high-level code to a single machine instruction, executed in a single clock cycle.

4.2.6.3 Header Field Operations

In the interest of minimizing packet header lengths, individual fields are usually encoded in the smallest number of bits possible, whether or not they overlap byte- or word-boundaries. This can be inconvenient for processors, particularly when a field overlaps two or more data words, as the Virtual Channel Identifier (VCI) field (shaded portion) would for a 16-bit processor in Figure 4.1. Two memory accesses and some awkward manipulation are required to fit the field into a register so that it can be analyzed. Dedicated discrete logic to align header fields, particularly if the protocol (and therefore header format) is known at design time, is cheap to implement, and can save a lot of cycles.

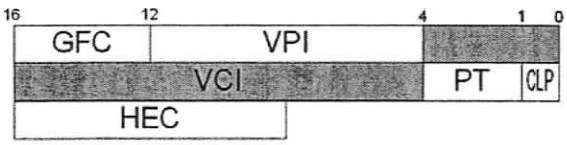

Figure 4.1: ATM Header Format

Analyzing fields like Payload Type (PT) in Figure 4.1 presents some difficulty, as well, because most processors do not have a 3-bit data type. Simple "greater than," "less than," or "equal to" comparisons are not useful, because when loaded as either an 8- or 16-bit value, the most significant bits are irrelevant to the comparison. It is useful in such cases to have either bit field operations or bit mask comparisons, or both, available to the programmer. For both non-aligned accesses, and masked operations, CISC machines are typically more robust and capable than most RISCs, and can accomplish these tasks more efficiently.

4.2.6.4 Checksums

Most protocols include some form of bitwise, packet level error detection, either for the payload or the header or both. The two most common types are simple arithmetic checksums and slightly more complex CRCs. The operations involved in the error detection are quite simple, but tedious over the length of the entire packet (often multiple times as one strips away protocol stack layers). Optimization or automatic calculation is inexpensive, in terms of added design and manufacturing to a processor. At a minimum, an accumulator (or a multiply-and-accumulate operation, if it operates quickly enough) can be advantageous.

4.2.7 Interrupts

Commonly, a computer system with a high bandwidth data link sends and receives data packets through a dedicated NIC over a media access control protocol like Ethernet. The NIC is responsible for processing the Ethernet packet layer, and passes up the payload, a transport-layer packet itself, to the processor by storing the packet in memory. A hardware exception is triggered, interrupting the CPU's regular processing and flushing

all instructions in the pipeline. Systems with lower bandwidth connections employ simpler protocols, the network interface is still likely to be interrupt-driven. Exceptions, or traps, are an inefficient use of processor cycles, but this mechanism worked well in the early days of networking when data arrived infrequently. With increasing network speeds and workloads, however, this technique limits a system's network bandwidth [90].

If a processor handles interrupts imprecisely, that is, without discarding instructions fetched before the interrupt occurs, additional latency is incurred while clearing those instructions. On the other hand, if an architecture handles interrupts precisely by discarding in flight instructions whenever an interrupt occurs, inefficiencies result when those instructions must be re-processed. Longer pipelines or long instruction resolution times (as in some special instructions, e.g., floating point multiplications, table lookups, etc.) exacerbate the issue. The cache effects mentioned in Section 4.2.2 are also contentious issues.

4.2.7.1 Timers

A considerable amount of the processing involved in networking relates to flow control and sequencing of incoming packets. This is especially true at the data link layer, for which an embedded system may not have external dedicated hardware (desktop computers usually do). As discussed in section 4.2.2, the availability of sufficient buffer space relieves the processor somewhat. Another element which can alleviate some of the processing burden is the availability of dedicated timers, to notify the processor at appropriate intervals when a flow control or link management message is due. Real-time systems often include an abundance of hardware timers already, so tasking them for network activity is trivial.

4.3 Putting It Together

We have discussed a number of specific architectural features and their relevance to communications and networking at the processor level. For low-bandwidth applications,

it is most advantageous to have functional units and instruction sets that are suitable for low-level manipulations commonly found in communications: bit-manipulation, switching, and block memory transfers increase the efficiency of such processing considerably.

Feature	Section	I.MX1	PPC 750
Lots of Buffer Space	4.2.1	Y	N
Von Neumann*	4.2.1	N	N
CISC	4.2.1	N	N
No Voltage/Clock Scaling*	4.2.2	Y	Y
Flexible Cache	4.2.2.1	N	N
Cache Line Locking	4.2.2.1	Y	N
DMA	4.2.2.2	Y	&
Short, Predictable Pipeline	4.2.3	5-stage	N
Scalar	4.2.3	Y	N
No Multithread*	4.2.5	Y	Y
Block Memory Movement	4.2.6.1	&	&
Switch Statement / Table Branch	4.2.6.2	N	N
Bit Manipulation	4.2.6.3	Y	Y
MAC	4.2.6.4	Y	N
Fast Interrupts	4.2.7	Y	N
Timers	4.2.7.1	Y	Y

Table 4.1: List of Beneficial Features for Networking
* Applies mainly to high bandwidth applications
& = Not supported, but a partial substitute exists

For high bandwidth applications, we have shown that the processor itself is a barrier to bandwidth, and that within the processor, memory transfers are the biggest bottleneck. This chapter has emphasized decreasing latency of interrupt handling, and increasing memory transfer efficiency as solutions to the problem. Table 4.1 summarizes the discussion and lists the features covered (along with the section in which they were

covered.) The last two columns of the table refer to two commercially available processors which are examined in the two subsections below.

4.3.1 Case Study – i.MX1

To illustrate some of the points in the preceding discussion, we examine the i.MX1 processor [103] designed by Freescale Semiconductors. Based on an ARM core (ARM920T), the i.MX1 is targeted specifically at multimedia mobile computing platforms, such as palmtop computers and personal digital assistants (PDAs), as well as 3G cellular telephony.

The ARM9 processor family is a 32-bit RISC solution, with a 5-stage pipeline. "Fast" interrupts allow process switching with a latency of only 3 clock cycles, instead of the "normal" 8, and are vectored and prioritized, allowing great flexibility and agility for response to external stimuli. There are two dedicated hardware timers (plus a watchdog timer), and a real-time clock to aid with flow control and responsiveness.

The instruction set is classically RISC, with a few notable exceptions: post- or pre-increment and –decrement addressing is provided, allowing data to be moved in contiguous blocks with only one instruction per move. In addition, ARM9 includes the 16-bit Thumb architecture extension for greater memory density and lower power consumption. Both the Thumb and ARM9 instruction sets include bit field operations and a wealth of test, compare, and branch operations. The multimedia accelerator contains a Multiply and Accumulate (MAC) unit which can assist with checksum or CRC calculation. The architecture is Harvard, which we identified in Section 4.2.1 as overqualified for low-bandwidth communications; it was probably chosen in order to facilitate the multimedia applications served by the chip.

128 KB of SRAM is included, as L2 cache, and could be very useful for buffering of network data. The ability of the ARM9 processor core to provide true CPU-cache transparency means that the i.MX1 can freely poll network interface status registers without cache-miss penalties. There is a Memory Management Unit (MMU) and an SDRAM controller to expedite memory accesses on the 32-bit 100MHz memory bus.

There is also an on-chip DMA controller, interfaced through local register transactions enabling DMA operations to be enacted with very low processor overhead.

For direct connectivity to simple networks, there are four options for serial ports: Serial Peripheral Interface (SPI), Universal Asynchronous Receiver/Transmitter (UART), Universal Serial Bus (USB), as well as the more specialized Inter IC (I2C) bus. In addition, there are multiple parallel interfaces, though these are unlikely to be used for inter-system communications.

Finally, in order to drive home the point that the i.MX1 is designed for mobile processing, Motorola has included built in Bluetooth baseband functionality in the form of a high-speed serial interface designed specifically to communicate directly with Bluetooth radio frequency (RF) transceivers. A software protocol stack is still required. To drive the application layer, MPEG4 video support, a dedicated video bus, a pulse width modulation module, an analog signal processor, and an LCD screen controller are built in, as well as MPEG4 and MP3 media compression features.

The i.MX1 is a very capable and flexible processor which, in addition to supporting a number of multimedia and auxiliary functions, is capable of communicating in a wide variety of networks. The architecture contains many of the features we have identified as beneficial, and the selection of serial ports plus the inclusion of a DMA engine and SDRAM controller allow it to be used effectively in low to medium bandwidth networks.

4.3.2 Case Study – PowerPC 750

The PowerPC architecture is one that has made its way from the desktop computing world to the embedded market, and continues to be actively developed for embedded applications. The PowerPC 750 [104] is a 2-way superscalar (3-way including a speculated branch), pipelined RISC processor. As an older architecture that has been recycled mainly due to popularity with programmers, it has many architectural features which are not well suited for network applications.

The pipeline is nominally four stages deep, but some of the functional units (the floating point unit, and the load-store unit) require additional stages. Superscalarity,

achieved through reservation stations on the functional units, and register renaming buffers, add additional uncertainty to the execution time of each instruction. Additionally, interrupts are handled precisely, requiring all instructions currently fetched to be executed completely and before beginning the interrupt handling process. This model accounts for unnecessary latency and is unsuitable for high bandwidth networking.

The Harvard setup has independent caches, which is not ideal, though they are capable of line locking, early restart, and wrapped fetches which help to reduce latency. However, there is no DMA controller. There is only a "bus interface unit" which is capable of some basic burst accesses between memory and cache.

In all, the PowerPC 750 is an example of a contemporary embedded processor based on a legacy architecture which is inefficient for both high speed and low speed communications. Its biggest deficiencies are insufficient memory support and an irregular and unpredictable pipeline made worse by precise interrupt handling. The memory could be improved with a more flexible cache and by the addition of an L2 cache.

4.4 Chapter Conclusion

This chapter has exemplified the sort of qualitative analysis of hardware and application that a system designer must undertake in the early stages of hardware specification. Since many factors must be considered in such an analysis, generalizations must be made. However, once the most significant factors have been identified, quantitative investigations should be undertaken in order to ensure correct selection or development of hardware to suit the individual application. Such an investigation is presented in the next chapter.

5. Optimizing On-Chip Memory for Energy and Speed Performance

5.1 Introduction and Motivation

The previous chapter examined a broad selection of hardware tradeoffs with a single class of applications, networking, in mind. In this chapter, we reduce the scope somewhat and focus in greater detail on a single tradeoff. The empirical, quantitative analysis in the following sections is applicable to all applications.

Unlike general-purpose computing, embedded applications can be completely specified at system design time. As a result, embedded systems can greatly benefit from extensive design-time optimizations of both software and hardware: software can be optimized based on a thorough knowledge of the hardware platform, and hardware can be optimized to fit the demands of the application. The memory hierarchy is one of the most critical optimization variables, as the memory sub-system is a major contributor to the overall power consumption and execution speed of the whole system. For example, memory accesses accounted for almost 70% of the total execution time of the *gsm* benchmark [105] in one of our simulation studies using a 4 KB cache.

For any embedded System-on-Chip (SoC), external memory accesses that must use off-chip busses are much more expensive than internal memory accesses. The amount of on-chip memory is limited, since the silicon area is constrained in order to keep the manufacturing costs as well as the chip footprint under control. Therefore, it is important to utilize the on-chip memory as efficiently as possible. The most common choice for the on-chip memory is cache, because of its historic prevalence in general-purpose computing. From the perspectives of energy and speed performance, however, a scratchpad can be the better alternative. The most general solution is to use both the scratchpad and the cache provided that an on-chip memory budget is not exceeded.

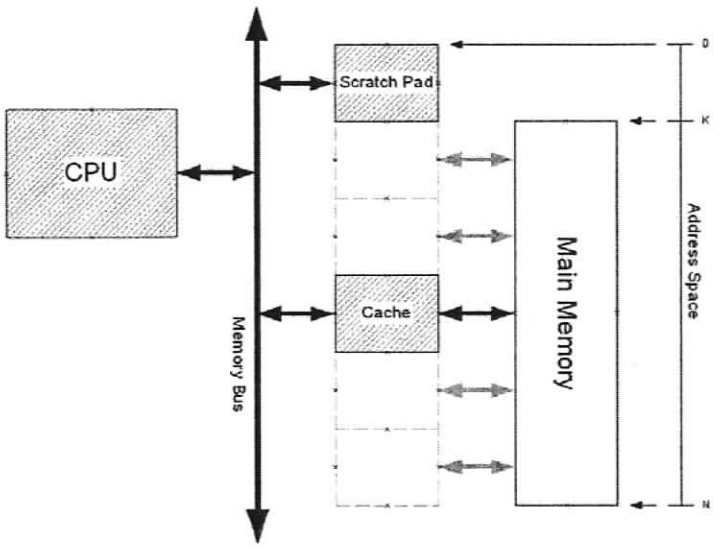

Figure 5.1: Memory configuration including both cache and scratchpad. Shading indicates those blocks which cohabit a single chip. Communications between these blocks are faster and more energy-efficient.

Figure 5.1 depicts a memory sub-system in which the memory map has been divided into main memory and scratchpad. The processor does not directly access the main memory, relying instead on the cache to pre-load portions of the working set for faster and more energy-efficient access. The pre-loading is dynamic and automatic; fetching and replacement are handled by logic that is integral to the cache design. The performance benefits (which will be discussed in Section 5.3.2) of scratchpad, in contrast, arise from the fact that the data and code in a scratchpad is static, and no such logic overhead is necessary. Objects are placed in the scratchpad explicitly by software, which means that special software routines or explicit instructions have to be included in code. Additional software can be added to enable replacement of objects to make a dynamic scratchpad, though the processing overhead involved in such replacement schemes is not necessarily efficient. This chapter deals primarily with static scratchpads into which objects are placed at startup and never removed or replaced. Both the scratchpad and the cache exhibit single-cycle access times, while accessing the external main memory is costlier in both latency and energy consumption.

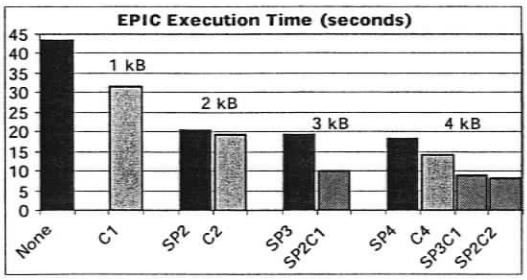

Figure 5.2: Execution times for *epic* benchmark with various memory configurations. The notation Cx denotes x KB of cache memory, while SPy denotes y KB of scratchpad memory.

An extensive amount of research has been done on the subject of efficient scratchpad utilization. The problem can be summarized as follows. Given a set of memory objects, either data or instructions, the goal is to map them selectively to the scratchpad memory space so that some performance metric (e.g., execution time, energy consumption) is optimized. Objects that are not mapped to the scratchpad reside in the off-chip main memory, whose access invokes a significant overhead in both latency and energy. One can use on-chip caches to reduce these penalties associated with off-chip memory traffic; however, care must be taken to reduce expensive cache misses.

The total area available for on-chip memory is often constrained by die and wafer geometry, and economic considerations. Given a fixed memory area on a SoC, one must decide how to partition the on-chip memory between the cache and the scratchpad. Different cache-scratchpad configurations can result in very different system performance values. For example, Figure 5.2 presents the execution times of the *epic* benchmark (see Section 5.4) with a variety of on-chip memory configurations[1]. The general trend shows that as on-chip memory is increased from 0 KB to 4 KB, the runtime decreases, but within each given memory budget, not all configurations perform equally well. The combined scratchpad- cache configurations in the 3 KB and 4 KB cases outperform both the all-scratchpad and all-cache variants.

---

[1] A greedy algorithm based on access frequency was used to pack the scratchpad with memory objects.

5.1.1 Our Contribution

Existing literature covers many aspects of managing the scratchpad memory space, both statically and dynamically, using complex compiler-controlled or hardware-controlled mechanisms[1]. In this chapter, we argue that comparable improvements in execution time and energy consumption can be achieved using a much simpler and more practical methodology, without modifying any compilers or application code, and without introducing any special hardware. Our approach is purely static, operating on both static data and code objects at link-time, employing only standard ready-to-use development tools and IP cores (e.g., CPU, scratchpad SRAM, cache with MMU). Our idea is very simple, yet it works very well:

1. First, we compile an application, trace its execution without any cache or scratchpad involved, and collect memory access statistics for static data and code objects. These objects are, in fact, ***.o*** files generated by the compiler without any interference, i.e., we maintain correspondence between the objects and the source code for easier system debugging during the design process.
2. Second, given an initial cache size (e.g., 4 KB), we run the application and record the value of our performance metric (e.g., energy, execution time). This is our first solution, with a scratchpad size of zero.
3. Third, we decrement the cache size by a certain amount (e.g., 1 KB), and increase the scratchpad size so that the total on-chip memory area (including the overhead of the cache control circuitry) remains approximately the same. We apply a simple yet efficient cache-aware heuristic, guided by the profiling information from step (1), to select objects for placement into the scratchpad, with the goal of minimizing off-chip memory accesses. We re-link the objects, according to their memory assignment, and re-run the application, obtaining different performance figures. This is our next solution, with the on-chip memory area budget partitioned between the scratchpad and the cache.

---

[1] A complete survey of existing literature is included in [106].

4. Fourth, we iteratively repeat step (3) until the cache size becomes zero, i.e., the scratchpad is the only on-chip memory. This is our last solution. Among all the generated solutions, we select the best per given performance metric for the application.

In other words, our approach explores both memory architecture and object assignment, subject to the on-chip memory size constraint. As our experimental results show, using our heuristic and changing the scratchpad-cache configuration may, in fact, improve application execution time and energy consumption on a par with complex object mapping schemes alone.

This chapter continues as follows: Section 5.2 introduces our techniques for deciding which memory objects are to be allocated to the scratchpad memory. Section 5.3 describes the experimental setup and in Section 5.4, our experimental results are presented. In Section 5.5, we explore variations of some finer architectural features of the on-chip memory (e.g., write-through vs. write-back caches) and discuss their impact on the system performance. Section 5.6 provides a detailed study of our scratchpad allocation heuristic, including the rationale behind it supported by the experimental data, and comparisons against optimal scratchpad allocations. Conclusions and future research directions are presented in Section 5.7.

5.2 Scratchpad Packing

Given an on-chip memory configuration with the corresponding fixed scratchpad size, our goal is to carefully select and place the most appropriate code and/or data objects into the scratchpad so that system performance is optimized. First, we sort the objects according to a certain priority metric. Then, the objects of this ordered list are assigned to the scratchpad in a first-fit manner respecting the size constraint.

5.2.1 Priority Metrics

The objects to be allocated to the scratchpad must be used with sufficient frequency as to justify their permanent inclusion in the fast memory. A simple and intuitive heuristic is to pack the scratchpad with the most frequently used objects. However, when cache is present, simply using the access frequency may not be as useful, since it does not account for the spread of the counted accesses over time. A low-frequency object left out of scratchpad, but whose access pattern exhibits poor temporal locality may cause frequent cache misses. It may be worthwhile to bring such an object into the scratchpad, replacing high-frequency objects with good temporal locality more efficiently handled by cache.

Therefore, in addition to the access frequency of an object, we need some measure of how tightly or widely spread those accesses are throughout the complete runtime. For examining quantitative statistics and qualitative characteristics of program objects, access profiles are generated by tracing the program's execution and associating each memory access with each object. The total runtime of the program was divided into evenly distributed "time bins," creating histograms similar to Figure 5.3 for each object.

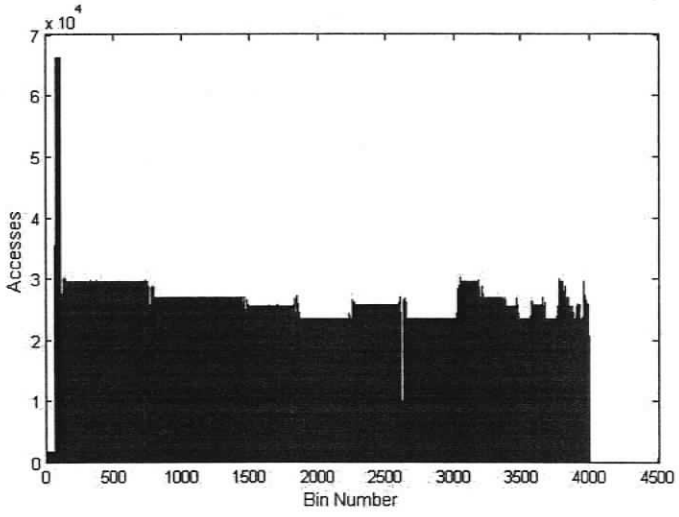

Figure 5.3: Sample time-access histogram used as an access profile for an object.

From the access profiles, we calculate the average access frequency **F** and the standard deviation, **σ**, of the memory reference distribution over time bins for a given object. It is important to differentiate between the average frequency, **F**, computed in the

y-axis, and the mean access time used in the calculation of **σ**, computed in the x-axis. A larger σ indicates that the access to the object are more widely spread throughout the total execution of the program. The product of **σ** and **F** provides a heuristic measure of the utility of servicing an object with scratchpad memory. In order to account for the cost of placing an object into the scratchpad, we divide the product of the standard deviation and access frequency by the size S of the object. Thus, for each object $\mathbf{o_i}$ in the set of all objects **O**, we calculate the priority metric

$$Sigma_i = \frac{F_i \times \sigma_i}{S_i} \qquad (\forall o_i \in O). \tag{5.1}$$

As an example, the right hand side of Table 5.1 shows the top twenty objects of the Sigma priority list for the *basicmath* benchmark from the MiBench [107]. For comparison, the left hand side lists the top twenty objects sorted by access frequency only, without taking the standard deviation or size into account. In both cases, items which would be included in a 4116 byte scratchpad are indicated with an X; there is a significant difference in the two packing strategies.

Frequency				Sigma			
Object Name	Type	Size (bytes)	SP4 (4116)	Object Name	Type	Size (bytes)	SP4 (4116)
dmul.o	Code	972	X	dmul.o	Code	972	X
daddsub.o	Code	1124	X	daddsub.o	Code	1124	X
ddiv.o	Code	1444	X	fputc.o	Code	44	X
flsbuf.o	Code	580		flsbuf.o	Code	580	X
dsqrt.o	Code	464	X	lludiv10.o	Code	152	X
__vfpntf.o	Code	3412		stdio.o	Data	204	X
pow.o	Code	4152		dsqrt.o	Code	464	X
_fp_disp.o	Code	2344		ddiv.o	Code	1444	
btod.o	Code	2200		libspace.o	Code	12	X
stdio.o	Data	204		printf.o	Code	84	X
lludiv10.o	Code	152		printf1.o	Code	4	X
jp_slve.o	Code	1208		rt_fp_status_addr.o	Code	16	X
poly.o	Code	348		ferror.o	Code	12	X
fputc.o	Code	44	X	istatus.o	Code	60	X
bigflt0.o	Code	348		poly.o	Code	348	X
basicmath_small.o	Code	2000		dflt.o	Code	100	
printf.o	Code	84		bigflt0.o	Code	348	
rt_sdiv.o	Code	212		sqrt.o	Code	112	
istatus.o	Code	60		abs.o	Code	12	X

Table 5.1: Priority list and scratchpad packing for the *basicmath* benchmark.

Intuitively, the proposed Sigma metric for a given object captures both its area efficiency (the number of access per byte of storage) and its inefficiency in terms of temporal locality (the spread of accesses over time). In other words, our metric does not favor large-sized object that can quickly exhaust available scratchpad space, but it does favor poor-locality objects to avoid cache misses. The latter point is what makes our heuristic *cache aware*. Complete details of our analysis of the Sigma metric are presented in Section 5.6.

5.2.2 Packing Algorithm

To select items to pack the scratchpad, we begin at the top of the priority list and add each object to the scratchpad if there is enough remaining space to do so. We continue in this manner until the scratchpad is full, or the end of the priority list has been reached. Our packing algorithm is shown below. Its space and time complexity is linear.

Scratchpad Packing Algorithm

Given an ordered list of k objects and a scratchpad of size x:

1. Set $i = 1$ and **spsize** $= x$.
2. If **size**$(o_i) \leq$ **spsize**, then
   2a. Add $\mathbf{o_i}$ to the scratchpad.
   2b. **spsize** = **spsize** – **size**(o_i).
3. If $i = k$ or **spsize** = 0 then terminate.
4. $i = i + 1$.
5. Go to 2.

5.3 Experimental Setup

To get the profile of an application and its objects' access frequencies and timing, we need a proper experimental setup for the tracing of program execution. After gathering the statistics of the executed application, we can then evaluate the optimal memory

configuration for a specific application. The purpose of our experiments is to expose the impact of scratchpad-cache partitioning on performance given a fixed on-chip memory budget.

5.3.1 Tracing the Application with ARMulator

Advanced RISC Machines (ARM) distributes an instruction set simulator, the ARMulator [108], which allows developers to benchmark ARM-targeted software on a variety of simulated platforms. To produce a target architecture for our simulations, we modified the ARMulator's model of the ARM710a processor: an ARM7 processor which includes a Memory Management Unit (MMU), a unified instruction/data cache, and an 8-entry, 4-address write buffer which we left unchanged. A direct-mapped, write-through, 4-word per line cache was used in simulation. We simulated the processor at 100 MHz, with a 100 MHz memory bus for the sake of simplicity. Off-chip memory was given a 100 ns access time, equating to 10 cycles per access, while on-chip SRAM accesses for both cache and scratchpad were conducted in a single cycle. For simplicity, and without a loss of generality, sequential and non-sequential reads and writes were assumed to have the same access time.

To prevent caching of the scratchpad, we took advantage of the ARMulator's ability to simulate the ARM710a memory paging system. By default, the lower 128 MB of the processor's memory map is cacheable and write-bufferable, and the remainder of the 4 GB address space is not cached. In our experiments, the scratchpad memory was mapped outside the lower 128 MB, and therefore not interfered with by the MMU.

The ARMulator's tracer was then enabled for the tracing of memory accesses, and the scatter-loading facility was used to separate all objects. While tracing, we executed an application with *neither cache nor scratchpad* in order to collect information about memory demand for each object without interference from the MMU. A trace of all accesses to memory, for the entire execution, and a map of all code and data objects in the memory as produced by the linker were obtained [109].

5.3.2 Memory Area and Energy

Cache memory has overhead in silicon area due to the tag array, comparators, and replacement logic. Without those elements, a scratchpad memory that occupies the same area has a larger capacity. [110] found a 34% area advantage for scratchpads, which we translated into an increase in capacity for the same area when trading between the two. We used the memory configurations in Table 5.2 for our experiments. The cache sizes chosen are typical of today's microprocessors.

As with area, and for the same reasons, cache consumes more energy per access than the scratchpad. The per-access energy figures for cache in Table 5.2 were obtained from the CACTI tool [59] for the direct-mapped, 16 byte-per-line cache described previously, implemented in a 0.13 μm process. The scratchpad values were also derived from CACTI, in the manner described by [110].

	Cache Size (bytes)	Cache Energy (nJ)	SP Size (bytes)	SP Energy (nJ)
SP0-C4	4096	2.35	0	N/A
SP2-C2	2048	1.82	2744	1.09
SP4-C1	1024	1.68	4116	1.34
SP5-C0	0	N/A	5488	1.59

Table 5.2: Memory Configurations and Energy Per Access

5.3.3 Performance Reporting

The ARMulator reports the total number of clock cycles for execution of a program. We multiply this cycle count by 10 ns (for a 100 MHz processor) to obtain the total execution time as one of our evaluation metrics. The ARMulator also reports the number of cycles spent accessing the cache and scratchpad memories. We multiplied those reported figures by the appropriate energy figures from Table 5.2 to measure on-chip energy consumption. This calculation included the on-chip portion of the penalty for cache misses in our energy comparisons. For off-chip memory (including the off-chip portion of the cache misses penalty,) we have averaged the per-access energy consumption of some commercially available SRAMs, such as the Cypress CYC1062AV33 [111], and arrived at the 6.13 nJ per access used in this chapter. In Section 5.5, we validate the use

of this estimated figure. The per-access cost was multiplied by the number of main memory accesses as reported by the ARMulator to calculate the energy consumption of off-chip memory.

In order to compare the relative energy efficiency of the memory configurations, we measured the total amount of energy consumed by the memory sub-system comprised of the cache, scratchpad, and off-chip memory (which also contains the stack and heap spaces for dynamic objects). This, along with the power dissipation of the processor and other peripherals, may be the primary consideration in battery-limited applications.

5.4 Experimental Results

5.4.1 The Benchmarks

In order to study the impact of our approach on the performance of a wide range of embedded systems, we chose benchmarks to represent a broad cross-section of typical embedded applications. The benchmarks were selected from the MiBench [107] and MediaBench [105] benchmark suites, and are described below:

epic An image compression utility developed at MIT, and intended for fast operation on integer hardware.

basicmath A solver of cubic functions, square roots, and angle conversions often found in control applications.

gsm A time and frequency division multiple access encoder of voice data.

stringsearch A case-insensitive search for short words in a large set of text strings.

dijkstra Dijkstra's algorithm to find the shortest path between every pair of nodes in a large adjacency graph.

patricia A simulator of routing computation using recorded IP traffic.

mpeg2enc A high-quality video encoder.

Figure 5.4 compares the memory sub-system activities of the seven benchmarks, and summarizes their average. The total number of data reads and writes are divided by the number of instruction reads to show the proportion of data activity to instruction activity.

On this scale, the instruction read rates for every benchmark would be unity. The cache activity rates, measured as the portion of total cycles in which a 4 KB cache is active (with no scratchpad present), are also shown.

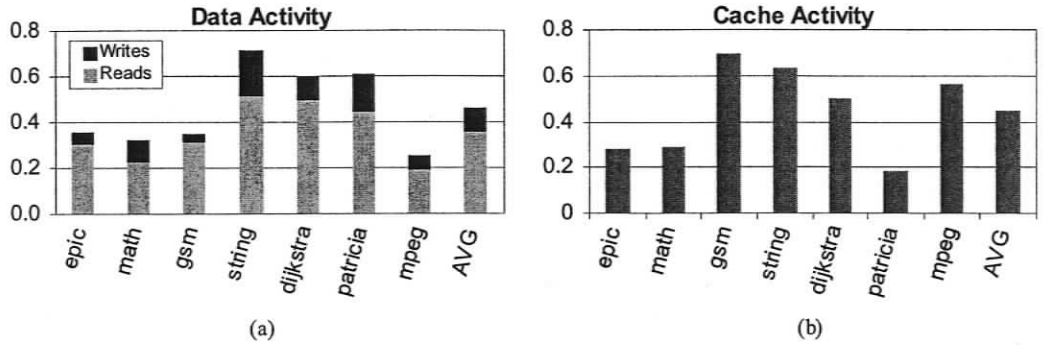

Figure 5.4: Memory activity in the seven benchmarks.
a) Data reads and writes are shown as a ratio of accesses per instruction. b) Cache activity is shown as a ratio of cache cycles to total cycles.

Both the *gsm* and *mpeg2enc* benchmarks show a small number of data accesses (similar to *basicmath* and *epic*) but a high degree of cache activity. We speculate that this is due to poor code locality. Conversely, over 60% of *patricia*'s instructions access data, and yet the cache activity is below average, leading us to believe that there is good locality in those data accesses.

5.4.2 Benchmark Results

Figure 5.5 plots runtimes and memory sub-system energy consumption normalized to the performance of the SP0-C4 cache-only case, alongside the range of variations across configurations. The configurations are ranked in Table 5.3, highlighting the most interesting result: no single memory variant consistently performs best or worst for all applications, for either measurement. In fact, each variant posts the best result for at least one benchmark. The all-cache case, which receives the worst overall ranking for energy efficiency, is the best choice for energy consumption when considering only the *dijkstra* benchmark (or equivalent workloads). This result supports our contention that optimization must be done on a case by case basis.

The runtime and energy consumption performances can range by as much as 80.3% and 155% (both for *gsm*) or as little as 4.7% (*patricia*) and 15.4% (*basicmath*), respectively. The chosen memory variant can be the dominating factor in system performance, far outstripping smaller optimizations achievable through sophisticated packing algorithms. Our performance reports include the contribution of unoptimized dynamic object accesses in addition to static data and instruction accesses optimized by our heuristic, i.e. the presented results show the complete memory cost.

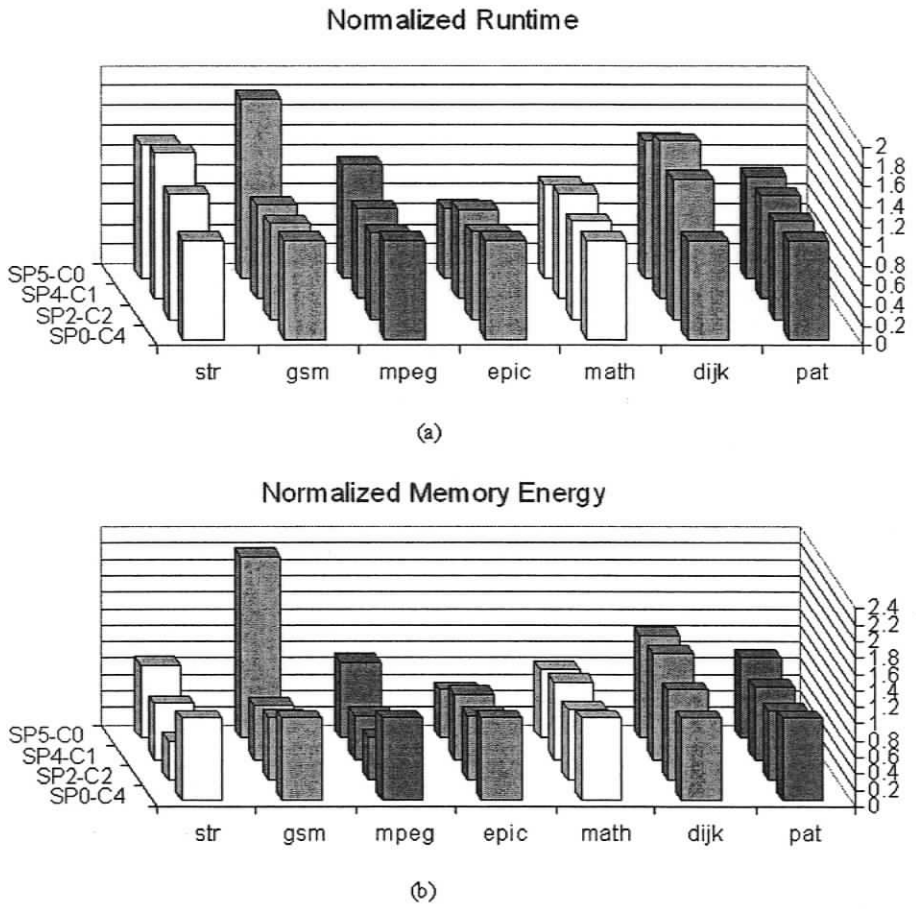

Figure 5.5: (a) Runtimes and (b) memory energy consumption normalized to SP0-C4.

Another valuable observation is that, while two of the benchmarks post identical rank orders for runtime and memory energy, five do not. This is an important fact for the system designer to keep in mind when choosing the system architecture. A text editor (as characterized by the *stringsearch* benchmark) in a wired system might choose the all-

scratchpad variant for best runtime efficiency, while a battery-powered device may get more value out of the mixed SP2-C2 combination.

	Runtime					Memory Energy				
	SP5-C0	SP4-C1	SP2-C2	SP0-C4	Range	SP5-C0	SP4-C1	SP2-C2	SP0-C4	Range
epic	**-1-**	3	2	4	29.4 %	**-1-**	3	2	4	42.3 %
math	**-1-**	4	2	3	12.0 %	**-1-**	3	2	4	15.4 %
gsm	4	**-1-**	2	3	80.3 %	4	**-1-**	2	3	155 %
string	3	4	2	**-1-**	46.8 %	3	2	**-1-**	4	53.1 %
dijk	2	4	3	**-1-**	58.1 %	3	4	2	**-1-**	25.1 %
pat	3	4	**-1-**	2	4.7 %	3	2	**-1-**	4	16.1 %
mpeg	4	2	**-1-**	3	26.3 %	3	2	**-1-**	4	49.0 %
Overall	3	4	**-1-**	2		3	2	**-1-**	4	

Table 5.3: Memory configuration rankings for runtime and energy consumption. 1 is best, 4 is worst.

It is meaningful to study the cache activity trends as well; basicmath, and epic, which showed low cache activity rates in the all-cache case (see Figure 5.4) and low data utilization rates, both perform very well with the all-scratchpad SP5-C0 variant, but very poorly with the all-cache SP0-C4. On the other hand, *gsm*, *mpeg2enc*, and *stringsearch*, which had the highest cache activities, do not run well on SP5-C0. Interestingly, all these benchmarks do not perform at their best with the all-cache variant either, preferring instead a heterogeneous structure.

5.5 Analysis Of The Experimental Architecture

The experimental architecture described in Section 5.3 and used to obtain the results in Section 5.4 is based on certain assumptions. In order to illustrate that the methodology presented in this chapter is valid for a wide range of architectures, we explore some of those assumptions in greater detail.

5.5.1 Main Memory Access Time

Our architecture imposed a 9-cycle penalty (for a total of 10 clock cycles) for accesses to off-chip memory accesses. However, there is a wide variety of memory in use in the

embedded design space, and a wide variety of processor clock frequencies. Experiments showed that runtime is indeed affected by the access penalty to main memory. Because the number of memory accesses to cache and scratchpad do not change, scratchpad-cache tradeoff is unaffected.

5.5.2 Cache Writeback Policy

We tested the benchmark *stringsearch* with a write-back cache to obtain the results in Figure 5.6. We observed a marked performance difference from both runtime and energy efficiency perspectives, including a 36% disparity in the energy consumption of the all-cache (SP0-C4) case. Experiments in changing the replacement policy from random to round-robin were less significant.

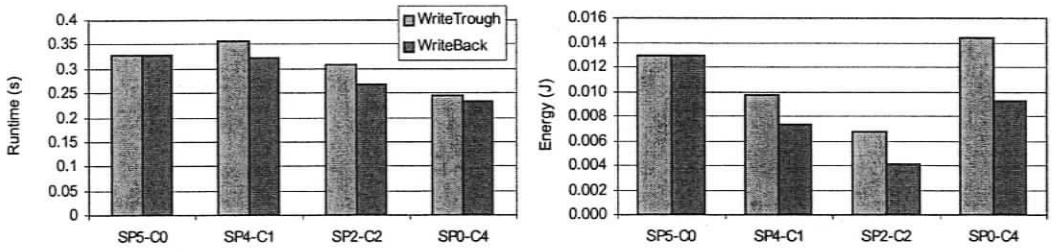

Figure 5.6: Execution time and memory energy consumption for *stringsearch* with write-through and write-back policies.

5.5.3 Cache Associativity

Figure 5.7 shows some measurements taken with 2-way and 4-way set associative caches. Runtime performance differences are small, with the greatest advantage observed in the SP0-C4 case where the 2-way cache outperforms the direct-mapped cache by 5.2%. There is noticeable increase in energy consumption as the associativity increases, due to a higher energy per access cost for caches of higher associativity (as obtained from CACTI).

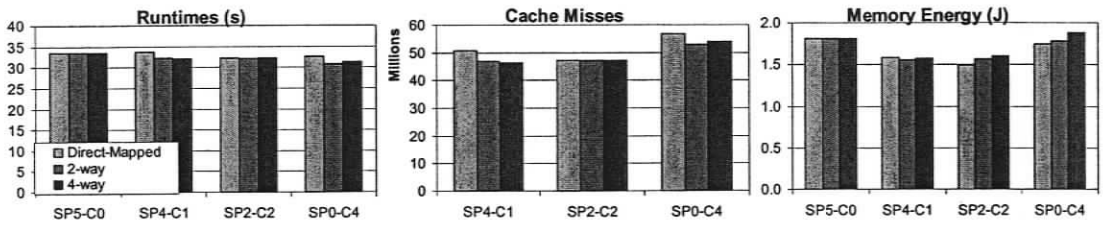

Figure 5.7: Runtimes, cache miss counts, and memory energy consumption for *patricia* with caches of varying associativity.

5.5.4 Cache Line Size

Figure 5.8 presents some measurements for the *mpeg2enc* benchmark with cache line sizes of 8, 16, and 32 bytes. As the line size increases, we observe a drastic drop in the number of cache misses, but an equally large increase in the number of accesses to off-chip memory, the net result of which is an overall increase in runtime for this benchmark. The amount of energy consumed by the memory sub-system increases in proportion.

5.5.5 Main Memory Access Energy

In real-world embedded systems, there is a great deal of variation in off-chip memory energy per access. The width, length, and number of the traces in the address and data busses, as well as the specific memory device chosen, and several other factors all impact the true cost. In order to demonstrate that our methodology is applicable to a variety of

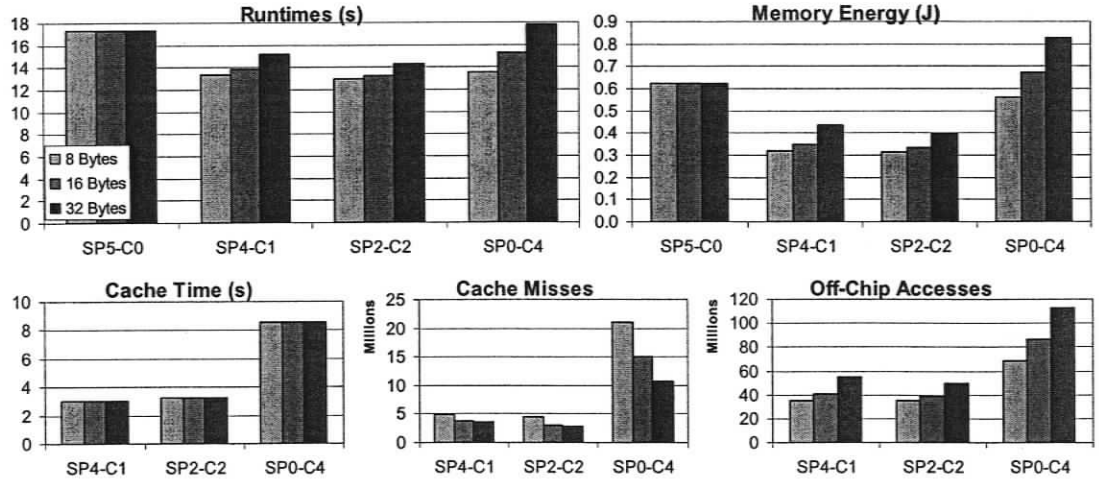

Figure 5.8: Measurements for varying cache line sizes for the *mpeg2enc* benchmark.

off-chip access costs, we present Figure 5.9 which shows the *dijkstra* memory energy consumption with an off-chip cost of 3.52 nJ (as opposed to the 6.13 nJ used everywhere else in this chapter), which was quoted from [112], and derived from a data sheet for RDRAM. The total energy consumed is dramatically reduced as expected.

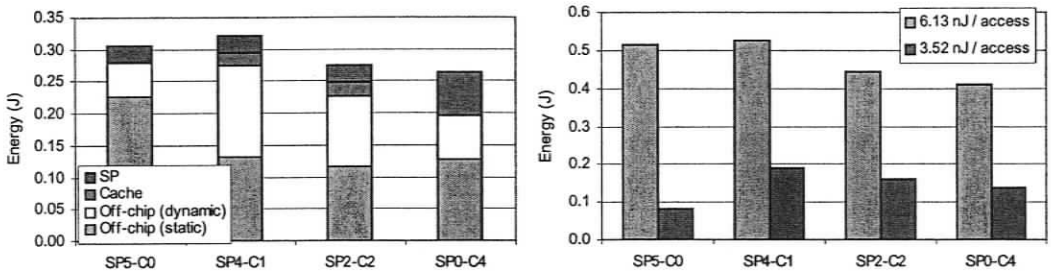

Figure 5.9: Memory sub-system energy consumption for the *dijkstra* benchmark with an off-chip memory cost of 3.52 nJ vs. 6.13 nJ per access.

5.5.6 Total On-Chip Memory Size

In the rest of this chapter, we have assumed a total on-chip memory area budget equivalent to a 4 KB cache. Figure 5.10 illustrates the scratchpad-cache tradeoff for the *stringsearch* benchmark in systems with an on-chip memory budget equivalent to 8 KB of cache. As with the 4 KB budget, the all-cache is the fastest and least energy efficient, while a heterogeneous variant is the most energy efficient.

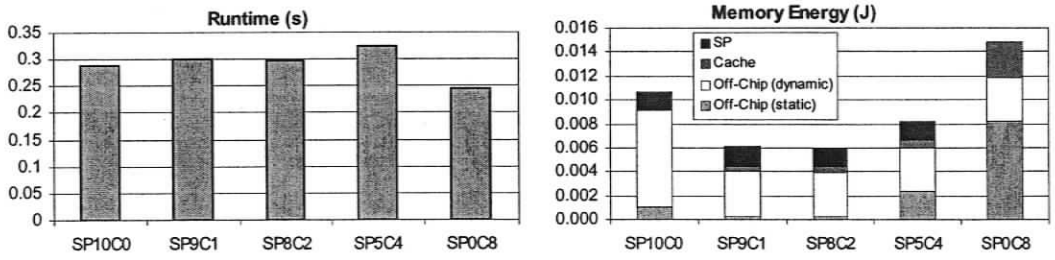

Figure 5.10: Performance of the *stringsearch* benchmark with 8 KB of on-chip memory space.

In some of these experiments (line size, access time, associativity) the architectural changes examined significantly affect performance, but do not change the relative performance of the memory variants in the cache-scratchpad tradeoff. In the other cases,

including the main memory access energy and the writeback policy, the relative performance of the memory variants is altered. While the methodology in this chapter remains applicable for all cases, these results emphasize the importance of a holistic approach to design space exploration.

5.6 Analysis of the Sigma Priority Metric

5.6.1 The Rationale for the Sigma Metric

The graphical representation of the access profile (as depicted in Figure 5.3) provides a perspective from which to interpret the access patterns intuitively. The area covered by the histogram is equal to the total number of memory accesses to that object. The shape of that area provides further insight: an evenly distributed graph indicates consistent use of that object, whereas a more varied topography indicates shifting demands. Thus, the histogram can help a designer gather insights into the execution of the software objects, and their demands on the memory sub-system, both cache and scratchpad.

Breaking average access frequency into total access count and number of bins, we dissect the formula in (5.1) to observe that access count and standard deviation (as defined in Section 5.2.1) are in the numerator, while runtime and size are in the denominator. For a given set of objects, the runtimes will be identical. Since there is no observed correlation between the shape of the access profile and either size or access count, we also hold those values constant in order to fairly compare potential access patterns. With these constants, a profile may vary in two ways, as demonstrated in Figure 5.11. As an object moves upward along the "Access Count" axis, bursts of activity become more intense, resulting in higher, tighter peaks. As an object moves forward along the "duration" axis, distinct bursts of activity to an object occur across larger periods of time.

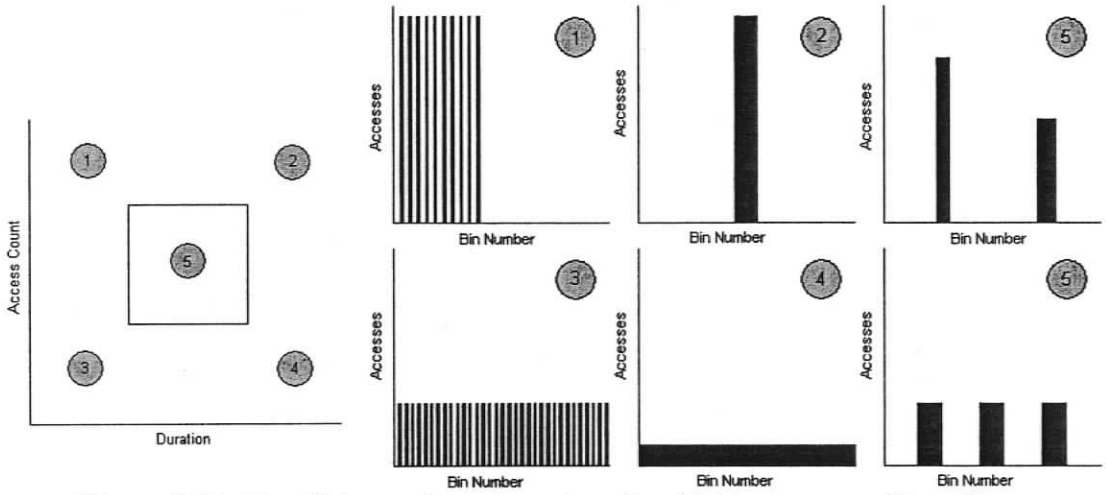

Figure 5.11: Identifying cache corner cases for objects access profiles. Size, total access count, and execution time are all held to be equal for all objects.

5.6.1.1 Cache-Efficient Objects

A cache is efficient when it does not have to replace its contents often. An object can help the cache be more efficient if its accesses are grouped tightly together, allowing less possibility for other objects to interfere. We observe that Type 2 in Figure 5.11 is the most advantageous for the cache, followed by Type 1, since both demonstrate sustained activity periods. Types 3 and 4 are both likely to lead to frequent replacement, but since Type 4 has the least access density possible, it should be considered the worst case scenario for cache support. Any other possibility, such as the two profiles labeled number 5, fall somewhere in between these others in terms of desirability. In order of preference for the cache, the types are ranked: 2,1,5,3,4.

5.6.1.2 Scratchpad-Efficient Objects

In our static scratchpad model, there is no dynamic fetching or replacement. Therefore, the scratchpad is oblivious to the pattern of accesses to an object, and is affected only by the volume of accesses, and the size of the object. By including the size in the denominator of the formula for the Sigma metric (5.1), we are effectively seeking to maximize the area-efficiency of the scratchpad construct in terms of operations per unit area in silicon. Since these two values are held constant for the objects in Figure 5.11, all objects are equally desirable, and we would expect any good metric to select objects for

scratchpad packing in order of inefficiency in the cache. Indeed we find that is the case; sorting the types by standard deviation from greatest to smallest yields the list: 4,3,5,1,2.

The above arguments illustrate standard deviation[1] as a factor in the Sigma metric. The other factors are intuitive: the average access frequency of an object directly dictates its effect on the performance of the memory sub-system, and linearly affects its contribution to scratchpad effectiveness. The size of an object is a measure of the cost of its inclusion into the scratchpad.

5.6.2 Scratchpad Efficiency

The great majority of objects found in the top ranks of the priority lists are code items, or instructions. Referring back to Table 5.1, we can see that for the *basicmath* benchmark, only one of the items in each list is a data object. In fact, out of the top 25 objects in the Sigma priority lists for each of the benchmarks, only 25% of the total size is attributable to data objects. In *stringsearch* there is one item sized 14780 bytes, at the 19th spot on the list. In *dijkstra*, the 17th-ranked item is 40824 bytes large. The great relative size of these two items to other code and data objects in the benchmarks distorts the general trend of code objects dominating the priority list, and at the same time prohibits their inclusion in even the largest of scratchpads. The data items that do make it into the scratchpad are likely to be smaller, on par with the size of the included code objects. We note that there is nothing in the Sigma metric which prefers code over data; this is a characteristic of the applications, not of the metric.

Figure 5.12 displays composites of the access profiles of all objects in the scratchpad for each benchmark. Each object included in the scratchpad is illustrated in a different shade and stacked with the others, so that the total darkened area represents all of the runtime memory accesses serviced by the scratchpad, or the scratchpad efficiency. The gaps

---

[1] It is important to note again that our standard deviation relates to the *times* of memory accesses, not to the number of accesses over time.

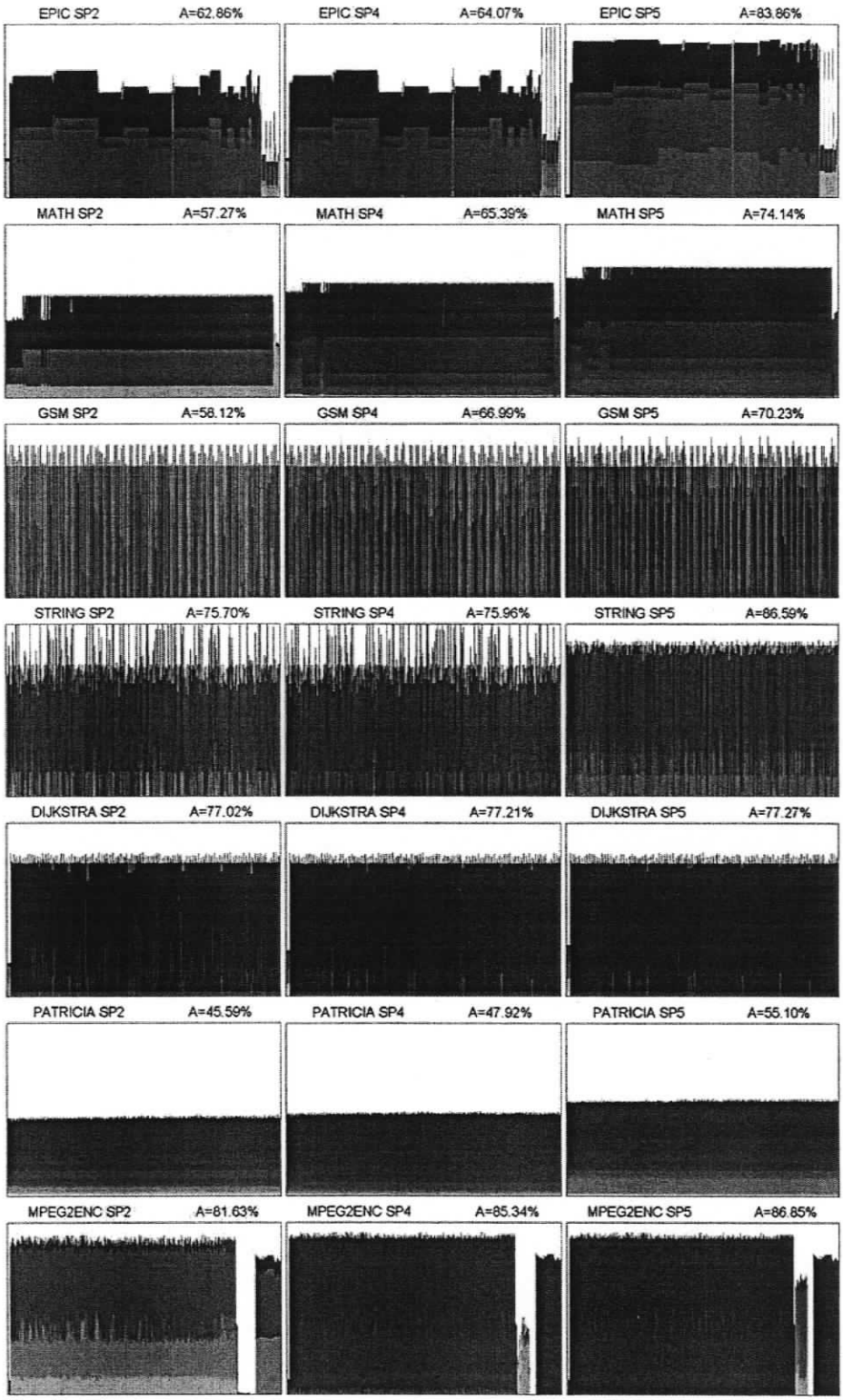

Figure 5.12: Scratchpad packing efficiency charts for seven benchmarks. The x-axes are the bin counts, and the y-axes are access counts per bin.

between dark areas represent the proportion of memory accesses which are handled by the cache.

A great deal of information can be gleaned from careful study of the efficiency charts. Horizontal bands or plateaus indicates a stable operating mode, wherein a program is continuously performing a single task, incurring accesses to objects uniformly over some period of time. A shift in the bands, as we see several times in *epic*, indicates a new task. Many of the benchmarks do not obviously exhibit this kind of shift; *stringsearch*, *gsm*, *patricia*, and *dijkstra* seem to have just one operating mode throughout their entire runtime. For a system processing an application similar to either of these benchmarks, it is likely that a statically-allocated scratchpad is preferable to a dynamic one, since the list of objects desired in the scratchpad does not change over time. For a more complex system involving multiple tasks and real-time user input, a dynamic solution may have merit, but at a cost of added uncertainty to real-time performance.

At first glance, we also detect a vertical striping in several of the benchmarks such as *gsm* and *stringsearch*. This striping implies a cyclic nature or repeated task, very likely due to a loop construct in the code. In fact, this small-scale repetition is present in all of the benchmarks, and is only obviously evident in the charts in those cases for which the time bin covers less time than the period of one repetition.

5.6.3 Impact of Scratchpad Efficiency on Performance

In many cases, we observe a direct correlation between the scratchpad efficiencies and the performances of the memory variants. Figure 5.12 shows that, for the *stringsearch* benchmark moving from SP2 to SP4, there is very little gain in scratchpad efficiency. Accordingly, the performance degrades as cache shrinks from C2 to C1 (see Figure 5.5). However, there is a large jump in scratchpad efficiency moving from SP4 to SP5, and the measured runtime improves, despite the complete removal of cache at that step.

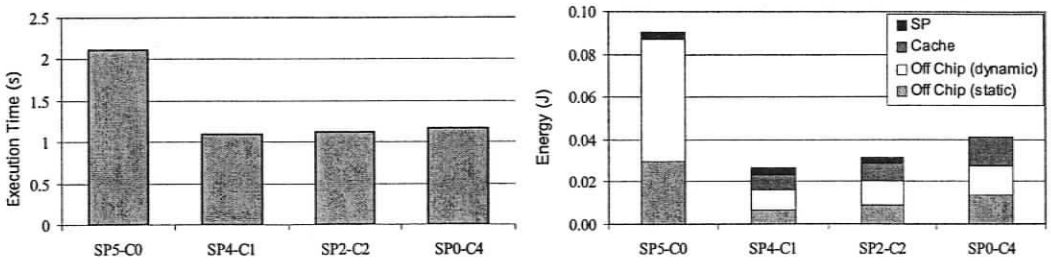

Figure 5.13: Execution time and memory sub-system energy consumption for *gsm*.

Figure 5.13 presents the simulated results for the *gsm* benchmark in greater detail. We can see that the all-scratchpad case is the least efficient in terms of both execution speed energy consumption. Referring to the packing efficiency charts for *gsm* in Figure 5.12, and the classification in Figure 5.11, we see that *gsm* has Type 1 gaps throughout its entire lifetime, meaning that there are Type 1 objects which were not included in the scratchpad. In the SP5-C0 case, without even a small cache to handle those objects, performance suffers considerably. For *gsm*, the SP4-C1 memory variant is the best option in both execution time and energy consumption.

Referring to Table 5.3 and Figure 5.5 for the *mpeg2enc* results measurements, we observe that the SP2-C2 configuration is the clear winner. From Figure 5.12, we see that *mpeg2enc* has a large Type 2 gap near the end of its run. We observe that as the scratchpad size increases from SP2 to SP5, the gap is not greatly reduced, and that even in the SP2 case, the remainder of the runtime is well-covered by the scratchpad. Recalling that a cache is quite well-suited to handling this Type 2 gap, it makes sense that the SP2-C2 variant performs better than the others for this benchmark.

In a case such as *dijkstra*, we observe from Figure 5.12 that the SP2 case has fair coverage, but the efficiency does not increase significantly as the scratchpad size grows, because the *dijkstra* objects, which are similar to the Type 3 pathological types in Figure 5.11, perform poorly in a cache, explaining the modest performance of the SP5-C0 case.

5.6.4 Effectiveness of the Scratchpad Packing Heuristic

Section 5.6.2 deals specifically with the effectiveness of the Sigma metric as a means of choosing static objects for scratchpad packing. The Sigma metric has several advantages, including linear time complexity and sensitivity to the presence of a cache, but it does not guarantee an optimal solution to the problem. It outperforms the simple access frequency metric, a somewhat falsely intuitive approach which ignores the impact of caching. To compare the two, we consider the SP4-C1 memory configuration, chosen to represent a case in which both cache and scratchpad are present. Figure 5.14a shows that Sigma is superior in all cases except for a 0.27% disadvantage in memory energy consumed for the *epic* benchmark. The average performance advantages are 3.9% and 10.4% for runtime and memory energy consumption, respectively.

To further evaluate the Sigma heuristic, we compare its packing results against an optimal packing solution when caching is not present. We focus our attention on the SP5-C0 configuration, where all static object accesses are made to either the scratchpad or the main memory, and therefore are free from interference from the cache. In this case, scratchpad packing can be cast as an instance of a classic knapsack problem [110] that can be solved optimally in pseudo-polynomial time by dynamic programming (DP) [113].

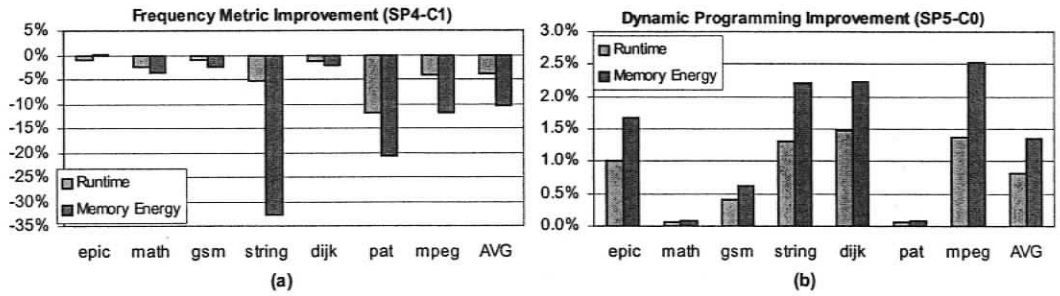

Figure 5.14: Performance gains of a) access frequency as priority and b) dynamic programming, relative to the Sigma metric.

Figure 5.14b compares the total execution time and memory energy consumption for two scratchpad packing cases for each benchmark: packing by the Sigma heuristic and packing by the optimal DP algorithm [114]. The SP5-C0 memory configuration was

chosen for these comparisons out of necessity; the DP solution becomes extremely complicated in the presence of a cache, because the true gain when placing an object in scratchpad is difficult to define. In all cases, the DP solution is superior, but the advantage is never greater than 1.49% (*dijkstra*) for execution time, and 2.53% (*mpeg2enc*) for memory energy. The difference in average runtime is only 0.81%, and the average memory energy consumption is only 1.35%. These results indicate that the performance of the Sigma heuristic measures very well against optimal packing solutions.

5.7 Chapter Conclusions

In this chapter, we describe a methodology for improving on-chip memory utilization by combining both scratchpad allocation and scratchpad-cache partitioning under a fixed budget for the on-chip memory area, i.e., without increasing the chip manufacturing costs. We consider not only all-scratchpad and all-cache configurations, but also various scratchpad-cache combinations. Using our approach a designer can achieve significant improvements in both application execution time (up to 80%) and energy consumption (up to 155%). We emphasize that our methodology is very simple, intuitive, immediately usable in practice, and fully platform-independent, with no custom software or hardware modifications required. We support this claim with a detailed quantitative study of several benchmarks, and expose the energy-speed tradeoffs not only at the software level (scratchpad allocation), but also at the hardware level (scratchpad-cache partitioning).

6. Conclusions

When developing custom hardware for embedded SoCs, it is vital to optimize the architecture to suit the final application. The complexity of such an undertaking is immense, but is simplified greatly by the modern paradigm of platform based design. Whether simply selecting hard IP cores, customizing soft IP, or developing fully custom blocks, careful attention must be paid to how each architectural decision affects the performance and efficiency of the system. This thesis has attempted to provide a basis for such considerations.

We present here a summary of the work contained in this thesis, as well as the opportunities that it presents for future work.

6.1 Contributions

This thesis makes several contributions to the field of embedded computing. In the first chapter we offer and defend a definition of the term "embedded" as it relates to the computing field. No such definition is currently universally accepted, leading to ambiguity amongst professionals. Additionally, we clarify terms including "platform based design," "performance," and "efficiency," in order to create a unambiguous basis for discussion in the chapters to follow.

In Chapter 2, we have developed a description of the types of workloads performed by embedded processors. We began at a fundamental level, describing the general goals and restrictions of embedded computing, before progressing into a discussion of the elemental units of work performed at the hardware level. The chapter concluded with a categorization of high-level workloads commonly performed in embedded systems. This is not comprehensive or rigorous, but serves to establish a language and structure for the discussion in Chapter 3.

Chapter 3 explores the architecture of embedded systems, including both general trends and specific features. Several tradeoffs are identified and discussed, with particular attention paid to the effects on system complexity, power efficiency, and processor performance. More importantly, individual architectural features are related directly to workload types and end applications. The discussion corresponding to each feature is intended to be of direct use to an embedded system designer seeking to customize hardware for a particular application. While the list is neither exhaustive in scope nor definitive in depth, to the best of our knowledge, it is the broadest treatment of the topic that exists in literature today, and can serve as a starting point for examinations of greater detail.

To support and illustrate the concepts in Chapter 3, we have completed two separate case studies, from two different perspectives. Chapter 4 focused on a single application and examined in greater depth the effects of a wide variety of architectural tradeoffs on hardware performance in that field. Chapter 5, on the other hand, focused on a single tradeoff. As the most rigorous chapter in this thesis, Chapter 5 makes several key contributions of its own.

First, it is an investigation of the area-constrained tradeoff of statically assigned scratchpad memory against cache, from both performance and power perspectives. The conclusion that this tradeoff has a varying optimal solution that is based on workload and operating constraints is important and consistent with the theme of this thesis. Second, the establishment of a profiling and optimizing procedure is useful as an iterative exploration that can be employed when choosing between many of the tradeoffs in this thesis. Thirdly and finally, the introduction of the Sigma metric as a means of statically assigning code and data objects to scratchpad memory is novel and practical, and of immediate usefulness in industry.

6.2 Future Work

It is our hope that the work laid out in this thesis will serve as the basis for future investigations; we identify several potential avenues for future research:

- Following the examination in Chapter 2, it would be beneficial to perform a rigorous, formal categorization of the elemental work units of embedded systems. A formal treatment may provide the basis for formal methods of optimization.
- The work in Chapter 3 should be extended to include a complete list of potentially beneficial architectural features. Such a list could be culled from historical and contemporary processors as well as academic literature, and could serve as a database for design methods of automated hardware generation or selection, especially in present-day CAD arenas.
- Chapter 4 discusses theoretical architectural choices for networking workloads in embedded processors, evaluates two commercially available solutions. It concludes that one processor is much more appropriate than the other, but neither are ideal. A custom-designed processor based on the recommendations in that chapter could be developed and benchmarked in order to demonstrate the methods and benefits.
- The work in Chapter 5 should be extended to include dynamic assignment of objects to the scratchpad, including dynamically generated "heap" objects.
- The methods in Chapter 5 have been simulated extensively, but should be evaluated in a real-world system; both real hardware and real applications should be investigated.
- Further experimentation with popular benchmark suites, and results analysis based on the Sigma heuristic would further refine packing strategies for various workloads.
- Relaxing the fixed area for on-chip memory and trading memory space against inclusion of other architectural features would allow another degree of freedom for optimization. Many such multi-feature tradeoffs should be examined.

References

[1] J. Turley, "The Two Percent Solution," Embedded.com. [Online]. December 2002. Avaliable: http://www.embedded.com/story/OEG20021217S0039.

[2] W. Wolf, "What is Embedded Computing," IEEE Computer, vol. 35, no. 1, January 2002, pp. 136-137.

[3] M. Schlett, "Trends in Embedded Microprocessor Design," IEEE Computer, vol. 31, no. 8, August 1998, pp. 44-49.

[4] S. Malik, M. Martonosi, and Y.-T. S. Li, "Static Timing Analysis of Embedded Software," Proceedings of the 34th Conference on Design Automation, June 1997, pp. 147-152.

[5] P. Petrov, and A. Oraloglu, "Customizable Embedded Processor Architectures," Proceedings of the Euromicro Symposium on Digital System Design, September 2003, pp. 468-475.

[6] D. Stepner, N. Rajan, and D. Hui, "Embedded Application Design Using a Real-Time OS," Proceedings of the 36th ACM/IEEE Conference on Design Automation, June 1999, pp. 151-156.

[7] J. Henkel, "Closing the SoC Design Gap," Computer, Vol. 36, No. 9, September 2003, pp. 119-121.

[8] PrimeXsys Platform Overview [Online] October 2005. Available: http://www.arm.com/products/solutions/PrimeXsysPlatforms.html

[9] Microchip Technology Inc. Website. [Online] October 2005. Available: http://www.microchip.com/

[10] L.Pillegi et al., "Exploring Regular Fabrics to Optimize the Performance-Cost Trade-Off," Proceedings of the 40th ACM/IEEE Design Automation Conference, June 2003, pp. 782–787.

[11] G. Martin and F. Schirrmeister, "A Design Chain for Embedded Systems," Computer, vol. 35, no. 3, March 2002, pp. 100-103.

[12] TI OMAP Homepage [Online] October 2005. Available: http://focus.ti.com/omap/docs/omaphomepage.tsp

[13] Intel Xscale Technology Overview [Online] October 2005. Available: http://www.intel.com/design/intelxscale/

[14] PowerPC Homepage [Online] October 2005. Available: http://www-03.ibm.com/chips/products/powerpc/

[15] Mips64 Homepage [Online] October 2005. Available: http://www.mips.com/content/Products/Cores/64-BitCores

[16] CoreConnect Bus Architecture [Online] October 2005. Available: http://www-306.ibm.com/chips/techlib/techlib.nsf/products/CoreConnect_Bus_Architecture

[17] AMBA Home Page [Online] October 2005. Available: http://www.arm.com/products/solutions/AMBAHomePage.html

[18] Altera Stratix Home Page [Online] October 2005. Available: http://www.altera.com/products/devices/stratix/stx-index.jsp

[19] Xilinx Virtex-4 FPGA Page [Online] June 2004. Available: http://www.xilinx.com/xlnx/xil_prodcat_landingpage.jsp?title=Virtex-4

[20] A. Sangiovanni-Vincentelli and G. Martin, "Platform-Based Design and Software Design Methodology for Embedded Systems," IEEE Design and Test of Computers, vol. 18, no. 6, November 2001, pp. 23-33.

[21] J. Turley, "The Two Percent Solution," *Embedded.com*. [Online]. December 2002. Avaliable: http://www.embedded.com/story/OEG20021217S0039.

[22] Y. Neuvo, "Cellular Phones as Embedded Systems," Solid-State Circuits Conference, 2004 (ISSCC). Digest of Technical Papers, February 2004, pp 32-37.

[23] Teoh King Long, Goh Mei Li, K. N. Seetharamu, and A.Y. Hassan, "A Fresh Look at Thermal Resistance in Electronic Packages," Electronics Packaging Technology Conference, 2000, December 2000, pp. 124 – 130.

[24] EEMBC Website. [online] October 2005. Available: http://www.eembc.com/

[25] D. A. Patterson, J. L. Hennessy, and D. Goldberg. "Computer Architecture : A Quantitative Approach," Morgan Kaufmann, 2nd edition, 1996.

[26] Y. Wu and J. R. Larus, "Static Branch Frequency and Program Profile Analysis," Proceedings of the 27th Annual International Symposium on Microarchitecture, November 1994, pp. 1-11.

[27] B. Calder, D. Grunwald, and A. Srivistava, "The Predictability of Branches in Libraries," Proceedings of the 28th International Symposium on Microarchitecture, November 1995, pp. 24-34.

[28] M. H. Miki, et al., "Evaluation of Processor Code Efficiency for Embedded Systems," Proceedings of the 15th International Conference on Supercomputing, June 2001, pp. 229-235.

[29] R. D. Silverman, "Exposing the Mythical MIPS Year," Computer, vol. 32, no. 8, August 1999, pp. 22-26.

[30] M. J. Flynn, "Very High-Speed Computing Systems," Proceedings of the IEEE, vol. 54, no. 12, December 1966, pp. 1901-1909.

[31] AMBA Home Page, [Online] October 2005. Available at: http://www.arm.com/products/solutions/AMBAHomePage.html.

[32] CoreConnect Bus Architecture Web Page, [Online] October 2005. Available at: http://www-03.ibm.com/chips/products/coreconnect/

[33] PCI-SIG Home Page. [Online.] Available at: http://www.pcisig.com/home

[34] C. Rijk, "Niagara: A Torrent of Threads," Ace's Hardware, April 2004. [Online] October 2005. Available: http://www.aceshardware.com/read.jsp?id=65000293

[35] CAN Remote Data Request Datasheet, EEMBC.org. [online] October 2005. Available: http://www.eembc.com/TechLit/Datasheets/auto_can.pdf

[36] ANSI/IEEE, New York. IEEE Standard for Binary Floating Point Arithmetic, Std 754-1985 edition, 1985.

[37] E. Kilgariff and R. Fernando, "The GeForce 6 GPU Architecture," NVIDIA website, 2005. [online] Available: http://download.nvidia.com/developer/GPU_Gems_2/GPU_Gems2_ch30.pdf.

[38] "Microprocessor Quick Reference Guide," Intel website. [online] October 2005. Available: http://www.intel.com/pressroom/kits/quickreffam.htm.

[39] I. Defee, "Software Decoding of HDTV," IEEE Transactions on Consumer Electronics, vol. 45, no. 4, November 1999, pp. 1277-1283.

[40] M. Arai, H. Arai, A. Kubota, and T. Sumi, "Adjustment-Free LSI Chipset for CD Player Application," IEEE Transactions on Consumer Applications, vol. 41, no. 3, August 1995, pp. 815-820.

[41] J. Huisken, "Components for Hand-Held Multimedia Devices," IEEE 3rd Workshop on Multimedia Signal Processing, Copenhagen, Denmark, September, 1999, pp. 533-534.

[42] J. Bustos and A. Bassi, "Voice compression systems for wireless telephony," Proceedings of the 21st International Conference of the Chilean Computer Society, November 1999, pp. 41-49.

[43] ARM11 Core Family Website. [Online.] October 2005. Available at: http://www.arm.com/products/CPUs/families/ARM11Family.html

[44] "Microcontroller Ideas Take Hold In Embedded Cores," Electronics Systems and Software, Vol. 1, No. 4, August 2003.

[45] J. Turley, "Embedded Processors, Part One," ExtremeTech website, January 11, 2002, [Online.] Available at: http://www.extremetech.com/article2/0,1697,18917,00.asp

[46] ARM Ltd. Webpage. [Online] October 2005. Available at: http://www.arm.com

[47] MIPS Technologies Webpage. [Online.] October 2005. Available at: http://www.mips.com/

[48] ARC International Webpage. [Online] October 2005. Available at: http://www.arc.com

[49] SuperH, Inc. Webpage, [Online.] October 2005. Available at: http://www.superh.com/home/index.htm

[50] ARM Press Release, June 16, 2003, [Online.] Available at: http://www.arm.com/news/319.html.

[51] J. Turley, "ARM Hammers on Thumb-2," Jim Turley's Silicon Insider, June, 2003. [Online.] Available at: http://www.jimturley.com/si/Issue%2003x/Issue%2003x.htm#story3

[52] G. Ungerer, "Using Flash Memory with uClinux," October 2002. [Online.] Available at: http://docs.linux.com/article.pl?sid=02/10/08/1912218

[53] N. Wirth, "Microprocessor Architectures: A Comparison Based On Code Generation By Compilers," Communications of the ACM, vol. 29, no. 10, October, 1986, pp. 978-990.

[54] "Embedded DRAM Market to Grow 21.3% - High-End Communications to Dominate Consumption," In-Stat.com, May 2002. [Online.] Available at: http://www.instat.com/press.asp?ID=198&sku=IN020103SI

[55] G. Harling, "Embedded DRAM Has a Home in the Network Processing World," EETimes.com, August 2001. [Online.] Available at http://www.eetimes.com/isd/features/OEG20010803S0026

[56] C.S. Wang and E.C.K. Chem, "Embedded-DRAM Technologies: Comparisons and Trade-offs," EDN.com, [Online.] September 2000. Available at: http://www.edn.com/article/CA47234.html

[57] P. J. Koopman Jr., "Stack Computers: The New Wave," Ellis Horwood Ltd., Chinchester, West Sussex, England, 1989. T. Lindholm and F. Yellin, "The Java Virtual Machine Specification," Sun Microsystems Inc., Palo Alto, CA, USA, 2nd edition, 1999.

[58] Jazelle DBX Overview, ARM Ltd. [Online] October 2005. http://www.arm.com/products/solutions/JazelleHardware.html

[59] S. J. E. Wilton and N. P. Jouppi, "CACTI: An Enhanced Cache Access and Cycle Time Model," IEEE Journal of Solid-State Circuits, vol. 31, no. 5, May 1996, pp 677-688.

[60] M. Campoy, A. P. Ivars, and J. V. Busquets-Mataix, "Static Use of Locking Caches in Multitask Pre-Emptive Real-Time Embedded Systems," Proceedings of IEEE/IEE Real-Time Embedded Systems Workshop, May 2001, pp. 1283-1286.

[61] W.-C. Hsu and J. E. Smith, "A Performance Study of Instruction Cache Prefetching Methods," IEEE Transactions on Computers, vol. 47, no. 5, May 1998, pp. 497-508.

[62] D. Katz and R. Gentile, "The Best Way to Move Multimedia Data," Embedded.com, December 2003, [Online.] Available at: http://www.embedded.com/showArticle.jhtml?articleID=16700107

[63] D. A. Patterson, J. L. Hennessy, and J.R. Larus. "Computer Organization and Design: The Hardware / Software Inteface," Morgan Kaufmann, 2nd edition, 1998.

[64] R. T. Short and H. M. Levy, "A Simulation Study of Two-Level Caches," Proceedings of the 15th Annual International Symposium on Computer Architecture, June 1988, pp. 81-88.

[65] N. P. Jouppi and S. J. E. Wilton, "Tradeoffs in Two-Level On-Chip Caching," Proceedings of the 21st International Symposium on Computer Architecture, April 1994, pp. 34-45.

[66] R. Hundal and V. G. Oklobdzija, "Determination of Optimal Sizes for a First and Second Level SRAM-DRAM On-Chip Cache Combination," Proceedings of the IEEE International Conference on Computer Design, October 1994, pp. 60-64.

[67] J. Kin, M. Gupta, and W. Mangione-Smith, "The Filter Cache: An Energy Efficient Memory Structure," International Workshop on Compiler and Architecture Support for Embedded Computing Systems, December 1997, pp. 184-193.

[68] L.H. Lee, B. Moyer, and J. Arends, "Instruction Fetch Energy Reduction Using Loop Caches For Embedded Applications with Small Tight Loops," Proceedings of the 1999 International Symposium on Low Power Electronics and Design, August, 1999, pp. 267-269.

[69] N. Bellas, I. Hajj, C. Polychronopoulos, and G. Stamoulis, "Energy and Performance Improvements in Microprocessor Design Using a Loop Cache," International Conference on Computer Design, October 1999, pp. 378-383.

[70] E. A. Lee, "Absolutely Positively on Time: What Would It Take?," IEEE Computer, vol. 38, no. 7, July 2005, pp. 85-87.

[71] S. Wichman and N. Goel, "The second generation ZSP DSP," LSI Logic Corp. June, 2004. [Online.] Available at: http://www.zsp.com/docs/pdf/technology/mpf2001_G2.pdf

[72] D. W. Wall, "Limits of Instruction Level Parallelism," Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems, April 1991, pp. 176-188.

[73] A. Uht, V. Sindagi, and S. Somanthan, "Branch Effect Reduction Techniques," IEEE Computer Magazine, vol. 30, no. 5, pp. 78-81, May, 1997.

[74] T.-Y. Yeh and Y. M. Patt, "A Comparison of Branch Predictors that Use Two Levels of Branch History," in Proceedings of the International Symposium on Computer Architecture, May 1993, pp. 257-266.

[75] S. McFarling, "Combining branch predictors," Digital Equipment Corporation Technical Note, DEC WRL TN-36, June 1993. [Online.] Available at: http://citeseer.ist.psu.edu/mcfarling93combining.html

[76] G. S. Tyson, "The Effects of Predicated Execution on Branch Prediction," Proceedings of the 27th International Symposium on Microarchitecture, December 1994, pp. 196-206.

[77] C. G. Lee and D. J. DeVries, "Initial Results on the Performance and Cost of Vector Microprocessors," Proceedings of the 30th Annual International Symposium on Microarchitecture, December 1997, pp. 171-182.

[78] R. Bhargava, L. K. John, B. L. Evans and R. Radhakrishnan, "Evaluating MMX Technology Using DSP and Multimedia Applications," Proceedings of the 31st IEEE International Symposium on Microarchitecture, December 1998, pp. 37-46.

[79] J. Tyler, J. Lent, A. Mather, and H. Nguyen. "Altivec: Bringing Vector Technology to the PowerPC Processor Family," Proceedings of the 1999 IEEE International Performance, Computing, and Communications Conference, February 1999, pp. 437-444.

[80] D. Geer, "Taking the Graphics Processor Beyond Graphics," IEEE Computer, vol. 38, no. 9, September 2005, pp. 14-16.

[81] D. Manocha, "General-Purpose Computations Using Graphics Processors," IEEE Computer, vol. 38, no. 8, August 2005, pp. 85-88.

[82] J. Stokes, "Introduction to Multithreading, Superthreading, and Hyperthreading," Ars Technica, October 2002. [Online] Available at: http://arstechnica.com/articles/paedia/cpu/hyperthreading.ars

[83] IA-32 Intel Architecture Software Developer's Manual, Intel Corporation, June 2005.

[84] J. Turley, "RISCy Business," Embedded.com, February 2003. [Online.] Available at http://www.embedded.com/story/OEG20030205S0025

[85] Tensilica Home Website, [Online] October 2005. http://www.tensilica.com.

[86] M. Moudgill and S. Vassiliadis, "Precise Interrupts," in IEEE Micro, vol. 16, no. 1, February 1996, pp. 58-67.

[87] R. D. Poor, "Wireless Embedded Networks: Connectivity for Everyday Objects," International Conference on Consumer Electronics, January 2005, pp. 145-146.

[88] "Typical Cellphone Data Speeds," Cellular-News.com, [Online.] October, 2005. Available at: http://www.cellular-news.com/3G/expected_speeds.php

[89] E. P. Markatos, "Speeding Up TCP/IP: Faster Processors are Not Enough," Technical Report TR297, Institute of Computer Science, Foundation for Research and Technology – Hellas, December 2001.

[90] T. Henriksson, U. Nordqvist, and D. Liu, "Embedded Protocol Processor for Fast and Efficient Packet Reception," 2002 IEEE International Conference on Computer Design: VLSI in Computers and Processors, September 2002, pp. 414-419.

[91] R. Arpaci-Dusseau, et al, "The Architectural Costs of Streaming I/O: A Comparison of Workstations, Clusters, and SMPs," Proceedings of the 4th International Symposium on High-Performance Computer Architecture, February 1998, pp. 90-101.

[92] NIST IEEE 1588 Website, [Online.] October 2005. Available at: http://ieee1588.nist.gov/

[93] L. Gwennap, "The Basics of Network Processors," Embedded.com, May 2004. [Online.] Available at: http://www.embedded.com//showArticle.jhtml?articleID=26806189

[94] S. Mukherjee and M. Hill, "Making Network Interfaces Less Peripheral," IEEE Computer, vol. 31, no. 10, October 1998, pp. 70-76.

[95] S. Mukherjee and M. Hill, "The Impact of Data Transfer and Buffering Alternatives on Network Interface Design," 4th International Symposium on High-Performance Computer Architecture, January 1998, pp. 207-218.

[96] D. Dunning and G. Regnier, "The Virtual Interface Architecture," Proceedings of Hot Interconnects V, ACM Press, August 1997, pp. 47-58.

[97] M. M. Mbaye, B. Tohio, Y. Savaria, and S. Pierre, "Performance of a Firewire-Ethernet Protocols Conversion on an ARM7 embedded Processor," Proceedings of the IEEE Canadian Conference on Electrical and Computer Engineering, 2003, vol. 2, May 2003, pp. 1267-1270.

[98] E. A. Lee and S. Neuendorffer, "Concurrent Models of Computation for Embedded Software," IEE Proceedings of Computers and Digital Techniques, vol. 152, no. 2, March 2005, pp. 239-250.

[99] H. Fallside and M. J. S. Smith, "Internet Connected FPGAs," 2000 IEEE Symposium on Field-Programmable Custom Computing Machines, April 2000, pp. 289-290.

[100] J. Pfrimmer, and K. F. Li, "Embedded 32-bit RISC IP Cores and OPEX Java Bytecode Folding," unpublished, University of Victoria, 2003.

[101] E. Cooper, O. Menziolcioglu, R. Sansom, and F. Bitz. "Host interface design for ATM LANs," Proceedings of the 16th Conference on Local Computer Networks, October 1991, pp 14-17.

[102] D. Tolley and S. Midkiff, "Analysis of CISC versus RISC microprocessors for FDDI network interfaces," Proceedings of the 16th Conference on Local Computer Networks, October 1991, pp 485-493.

[103] "i.MX1 Product Summary Page" Freescale Semiconductors [Online.] October 2005. Available at: http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=i.MX1

[104] PowerPC 750 Microprocessor Home Page, IBM Microelectronics, [Online.] October 2005. Available at http://www-306.ibm.com/chips/techlib/techlib.nsf/products/PowerPC_750_Microprocessor

[105] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A Tool For Evaluating and Synthesizing Multimedia and Communications Systems," Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture, December 1997, pp. 330-335.

[106] J. Pfrimmer, K. F. Li, and D. Rakhmatov, "Optimizing On-Chip Memory for Energy and Speed Performance in Embedded Systems," Technical Report ECE-05-3, Department of Electrical and Computer Engineering, University of Victoria, Victoria, Canada, October 2005.

[107] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," Proceedings of the 4th Annual IEEE International Workshop on Workload Characterization, December 2001, pp. 3-14.

[108] "Application Note 32: The ARMulator," ARM Limited, 2003. [Online.] Available at http://www.arm.com/pdfs/AppNote32_ARMulator.zip. Apr. 1999

[109] J. Pfrimmer, K. F. Li, and D. Rakhmatov, "Profiling Static Program Objects Using the ARMulator," Technical Report ECE-05-1, Department of Electrical and Computer Engineering, University of Victoria, Victoria, Canada, January 2005.

[110] R. Banakar, S. Stienke, B.-S. Lee, M. Balakrishnan, and P. Marwadel, "Scratchpad Memory: A Design Alternative For Cache On-Chip Memory In Embedded Systems," 10th International Symposium on Hardware/Software Codesign, May 2002, pp. 73-78.

[111] Cypress Semiconductor Corporation, "CY7C1062AV33 512Kx32 Static RAM," Document #38-05137. [Online.] February 2003. Available at http://www.cypress.com.

[112] V. Deleluz, M. Kandemir, N. Vijaykrishnan, and M. Irwin, "Energy-Oriented Compiler Optimizations For Partitioned Memory Architectures," Proceedings of the 2000 International Conference on Compilers, Architectures, and Synthesis for Embedded Systems, November 2000, pp. 138-147.

[113] S. Sahni, "Data Structures, Algorithms, and Applications in C++," McGraw-Hill, Boston, 1998.

[114] D. Rakhmatov, personal communications, January, 2005.

Appendix A. Profiling Static Program Objects

For ARM architectures, the ARMulator from ARM Systems Limited is a powerful tool for the simulation of ARM hardware and compiled software. This document will detail the steps and components necessary to collect memory usage data and generate statistics about static program and data objects in compiled software. Dynamic memory objects are not included, but similar methods may be applicable.

The ARMulator has its own built-in profiler, intended to provide much of the same information that is gathered using the steps in this document. However, the methods herein will yield information in much greater detail and accuracy. Further, as these methods deal in extraction and distillation of raw data, they may be extended and customized more readily than the ARM-included profiler.

This document assumes that the reader has a passing familiarity with the ARM Developer Suite (ADS) (specifically, version 1.1), and a working, standard installation of the same. The reader should have compiled and simulated the software to be profiled, using the ADS C compiler, linker, and ARMulator before beginning.

A.1 Separating Code and Data Objects

The ARMulator must be configured to employ a scatter-loading file similar to the one listed in Figure A.1, in order to separate the objects into their code, constant, and data components. The file breaks down as follows:

MAIN: This section is the loading region. It encompasses all following execution regions

CODE: Contains all read-only instruction objects

CONST: Contains all read-only data objects.

DATA: Contains all read-write data objects

ZI: Contains all zero-initialized read-write data objects.

Scatter.scf

```
MAIN      0x6FD0000
{
    CODE 0x6FD0000 0x010000
    {
        *(+CODE)
        anon$$obj.o(+CONST)
    }
    CONST 0x6FE0000 0x010000
    {
        *(+CONST)
    }
    DATA 0x6FF0000 0x008000
    {
        *(+RW-DATA)
    }
    ZI 0x6FF8000 0x008000
    {
        *(+ZI)
    }
```

Figure A.1: Scatter File for Object Separation

The "anon$$obj.o" name in the CODE area is to ensure that the data lookup tables are in the root region, as required by the ARM compiler. This does not cause any source code or data to be placed in the wrong region.

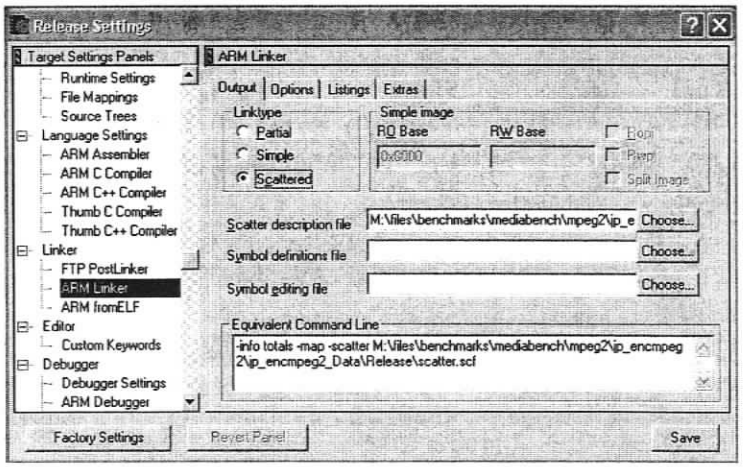

Figure A.2: Enabling Scatter Linking

Once this file has been created, scatter loading must be enabled within the ARM linker. This is done in the release settings (Edit -> Release Settings) as shown in Figure A.2.

When these steps are complete, compiling and linking the source code will produce a listing of the names, sizes, types, and locations of all objects in the memory map, provided the appropriate settings are enabled, as in Figure A.3. The Image map will be outputted to the "Errors and Warnings" window in CodeWarrior and should be saved to a text file.

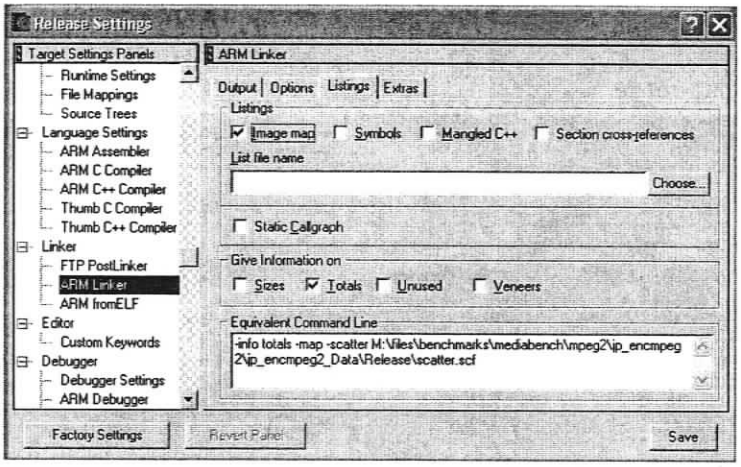

Figure A.3: Producing an Image Map

A.2 Creating a Cacheless Processor Variant

In order to collect memory access data without interference from the MMU, we generate a processor model which does not have an MMU. This is done by editing the file armulate.dsc (normally found within C:\Program Files\ARM\ADSv1_1\Bin\) and adding the bolded lines in Figure A.4.

This will create an option in the Release Settings Target Processor drop-down menu to select the processor for simulation, as in Figure A.5.

Armulate.dsc :

```
...
;; Variants of ARM7:
{ARM70=ARM7
Processor = ARM70
}

ARM700=ARM7
ARM704=ARM7
ARM710=ARM7
ARM710a=ARM7

Josh_C0=ARM7
Josh_C0:Processor=Josh_C0
Josh_C0:HASMMU=FALSE
```

Figure A.4: Editing armulate.dsc to create a processor variant

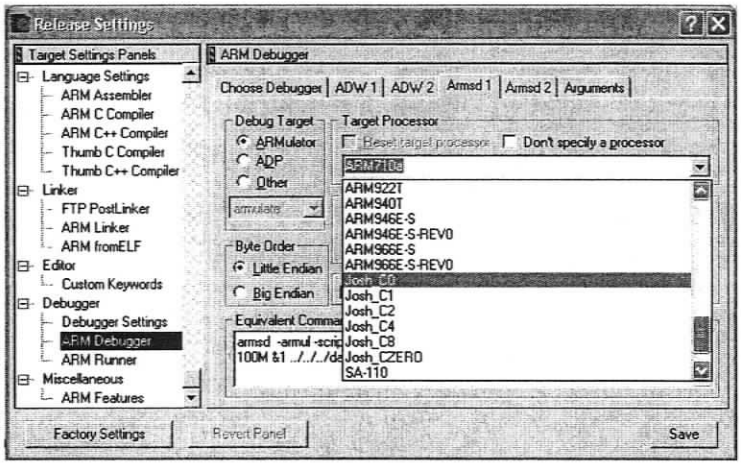

Figure A.5: Selecting the Processor Variant

A.3 Tracing the Execution

The ARMulator must be configured for tracing by editing the peripherals file (normally located in C:\Program Files\ARM\ADSv1_1\Bin\Peripherals.ami). The listing in Figure A.6 contains the tracer relevant section of *Peripherals.*ami and lists the correct setting to trace only the memory accesses.

Peripherals.ami :

```
{ Default_Tracer=Tracer
RDILog=False
File=d:\armul.trc
TraceInstructions=False
TraceRegisters=False

TraceMemory=True

TraceIdle=False
TraceNonAccounted=False
TraceEvents=False
TraceBus=False
Disassemble=False
StartOn=True
}
```

Figure A.6: Enabling the Tracer for Memory Accesses

The ARM Debugger equivalent command line is as follows:

```
armsd  -armul -cpu Josh_C0 -clock 100M &1
```

where Josh_C0 is a modified version of the ARM710A processor with the cache removed.

The output of this execution is a very large trace file listing each memory access over the course of program execution. To distil this information to a usable form, a Matlab script and C executable (Appendices B and C) are run with the trace and ARM Linker listing (from Section A.1) as input. This script also generates some useful statistics regarding the access patterns of the memory objects.

The script begins by reading in the link information text file and the names, sizes, and locations of each memory object. It then parses the trace file, line by line (memory access by memory access), in order to generate certain statistics, along with a time histogram of the memory accesses, which are useful as access profiles for characterizing the software's resource requirements.

Appendix B. Trace Parsing Matlab Script

```
function RANK = parsetrace(InputFile, LinkInfoFile, OutFile, XLSfile)
% ParseTrace - Distill ARMulator trace file
%
%  Usage: parsetrace InputFile LinkInfoFile OutFile XLSfile
%
%   This script accepts an ARMULATOR trace file and bins all memory
%   accesses.  The output is a tab-delimited text file suitable
%   for entry into a spreadsheet program.

if nargin ~= 4
   disp 'not enough arguments -- type "help parsetrace"'
   return
end

%set up some constants
mainsize = 196608;
%mainlowerlimit = 134021120;
mainlowerlimit = 83689472;
spadsize = 6144;
%sizeofbin = 4;
DeltaT = 100000;

disp 'Reading Linkinfo file';

%first, we need to come up with a list of objects
linkinfo = fopen(LinkInfoFile);
numobjects=0; while feof(linkinfo) == 0
    LINE = fgetl(linkinfo);

    %if it deosn't start with an address, skip the line
    if strncmp(strtok(LINE),'0x',2) == 0
        continue;
    end

    %now the string is an object listing of the form
  %BaseAddr    Size        Type  Attr Idx E Section Name
  %0x04fd0000  0x000000a8  Code  RO   32  * !!!
__main.o(c_a__un.1)
  %0x04fd00a8  0x00000848  Code  RO    1    .text   epic.o

    %pull off the start address and the size
    [OBJSTART, LINE] = strtok(LINE);
    OBJSTART = hex2dec(OBJSTART(3:end));
    [OBJSIZE, LINE] = strtok(LINE);
    OBJSIZE = (hex2dec(OBJSIZE(3:end)));
    OBJEND = OBJSIZE + OBJSTART;
```

```
        %find out what kind of object it is
        [OBJTYPE, LINE] = strtok(LINE);
        [OBJTYPE2, LINE] = strtok(LINE);
        if strcmp(OBJTYPE,'Code')
            SUFFIX = '(+CODE)';
        elseif (strcmp(OBJTYPE,'Data') && strcmp(OBJTYPE2,'RO'))
            SUFFIX = '(+CONST)';
        elseif (strcmp(OBJTYPE,'Data') && strcmp(OBJTYPE2,'RW'))
            SUFFIX = '(+RW-DATA)';
        elseif strcmp(OBJTYPE,'Zero')
            SUFFIX = '(+ZI)';
        else
            continue;
        end

        %grab the name of the object, and add the suffix
        %OBJNAME = [char(regexp(LINE,'\S*[.]o','match')),SUFFIX];
        [a,b] = regexp(LINE,'\S*[.]o','once');
        OBJNAME = [LINE(a:b),SUFFIX];

        %put it in the list
        numobjects=numobjects+1;
        OBJECTS{numobjects}.NAME = OBJNAME;
        OBJECTS{numobjects}.START = OBJSTART;
        OBJECTS{numobjects}.END = OBJEND;
        OBJECTS{numobjects}.SIZE = OBJSIZE;

    end %end linkinfo file loop
    fclose(linkinfo);

    %open file trace for editing
    trace = fopen(InputFile);
    %how big is it?
    fseek(trace,0,'eof');
    tracefilesize = (((ftell(trace) - 80) / 28) + DeltaT);
    fclose(trace);

    for count = 1:numobjects
        OBJECTS{count}.total = 0;
        OBJECTS{count}.mean = double(0);
        RANK(count) = OBJECTS{count};
    end

    disp 'Calling C routine to read trace';

    readtrace(InputFile,tracefilesize,RANK,OutFile);

    disp 'Calculating Statistics'
    load(OutFile);

    for count = 1:numobjects
        if mod(count,10) == 0
            fprintf(1,'%2.2f percent complete\r',(count /
numobjects)*100);
        end
        if OBJECTS(count).total > 0
```

```
            %OBJECTS{count}.std_dev = ...
             std(OBJECTS{count}.TIMES(1:OBJECTS{count}.total));
            OBJECTS(count).std_dev = 0;
            OBJECTS(count).mean = (OBJECTS(count).mean / ...
                                              OBJECTS(count).total);
            for count2 = 1:(numel(OBJECTS(count).ACCESS))
                OBJECTS(count).std_dev = OBJECTS(count).std_dev + ...
                   (OBJECTS(count).ACCESS(count2) * (((count2*DeltaT) -
...
                                                 OBJECTS(count).mean)^2));
            end
            OBJECTS(count).std_dev = sqrt( OBJECTS(count).std_dev / ...
                                                   OBJECTS(count).total);
        else
            OBJECTS(count).std_dev = inf;
        end
        OBJECTS(count).APB = OBJECTS(count).total / OBJECTS(count).SIZE;
    end

    disp 'sorting'

    %make a list of all the std deviations
    for count = 1:numobjects
            STDS(count) = OBJECTS(count).std_dev;
    end

    clear RANK;

    for count = numobjects:-1:1
        [m,t] = max(STDS);
        %remove the max one
        STDS = [STDS(1:(t-1)),STDS((t+1):count)];
        RANK(count) = OBJECTS(t);
        OBJECTS = [OBJECTS(1:(t-1)),OBJECTS((t+1):count)];
    end

    %save it to the outfile
    delete(OutFile);
    save(OutFile,'RANK');

    disp 'done!'
```

Appendix C. Readtrace.c Called by Matlab Script

```
#include "mex.h"
#include "string.h"
#include "mat.h"
#include <stdio.h>
#include <stdlib.h>

#define MAXLINELENGTH   100
#define DELTAT          100000
#define REPORTTIME      4000000
//#define MAINLOWERLIMIT  134021120
#define MAINLOWERLIMIT  83689472
#define MAINSIZE        196608
#define SPADSIZE        6144

//call with arguments
// filename
// tracefilesize
// OBJECTS array
// output filename
void mexFunction(int nlhs, mxArray *plhs[], int nrhs,
                 const mxArray *prhs[])
{
    char *LINE, *TYPE, *outfilename;
    int   buflen;
    FILE *trace;
    int time, tracefilesize;
    long ADDRESS;
    int count;
    int numobjects, numbins;
    int temp;
    double *pointer;
    double *pointer2;
    double **ACCESS;
    mxArray *OUTPUT;
    MATFile *obj;

    /* Check for proper number of arguments. */
    if (nrhs != 4) {
        mexErrMsgTxt("Four inputs required.");
    } else if (nlhs > 0) {
        mexErrMsgTxt("Too many output arguments");
    } else if (!mxIsStruct(prhs[2])) {
        mexErrMsgTxt("3rd input must be a structure.");
    } else if (mxIsChar(prhs[0]) != 1) {
```

```
        mexErrMsgTxt("1st Input must be a string.");
    } else if (mxIsChar(prhs[3]) != 1) {
        mexErrMsgTxt("4th Input must be a string.");
    } else if (mxGetM(prhs[0]) != 1) {
        mexErrMsgTxt("1st Input must be a row vector.");
    }

    tracefilesize = mxGetScalar(prhs[1]);
    numobjects = mxGetN(prhs[2]);
    numbins = tracefilesize / DELTAT;
    time = 0;

    /* Get the length of the input string. */
    buflen = (mxGetM(prhs[0]) * mxGetN(prhs[0])) + 1;
    /* Allocate memory for input and output strings. */
    LINE = mxCalloc(MAXLINELENGTH, sizeof(char));
    /* Copy the string data from prhs[0] into a C string
     * FileName. */
    mxGetString(prhs[0], LINE, buflen);
    /* Get the length of the input string. */
    buflen = (mxGetM(prhs[3]) * mxGetN(prhs[3])) + 1;

    /* Allocate memory for input and output strings. */
    outfilename = mxCalloc(MAXLINELENGTH, sizeof(char));
    /* Copy the string data from prhs[0] into a C string
     * FileName. */
    mxGetString(prhs[3], outfilename, buflen);

    //create a copy of the input array so we can work with it
    OUTPUT = mxDuplicateArray(prhs[2]);

    mexPrintf("Allocating memory\n");
    //allocate LOTS of memory for the ACCESS arrays
    ACCESS = mxCalloc(numobjects, sizeof(double *));
    for(count = 0;count < numobjects;count++)
    {
        ACCESS[count] = mxCalloc(numbins, sizeof(double));
        if (ACCESS[count] == NULL) mexErrMsgTxt("Not enough memory");
        for(temp = 0; temp < numbins; temp++) ACCESS[count][temp] =
0;
    }
    //check to see if there will be room to allocate the rest
    pointer = mxCalloc((numobjects * numbins),sizeof(mxREAL));
    if (pointer == NULL) mexErrMsgTxt("Not enough memory");
    mxFree(pointer);

    trace = fopen(LINE,"r");
    if (trace == NULL) mexErrMsgTxt("Couldn't open file");

    //file is now open, skip the first 4 lines
    fgets(LINE,MAXLINELENGTH,trace);fgets(LINE,MAXLINELENGTH,trace);
    fgets(LINE,MAXLINELENGTH,trace);fgets(LINE,MAXLINELENGTH,trace);
    mexPrintf("Reading trace file\n");

    //read in the trace, a line at a time
    //time=DELTAT; //so that the index is never zero
```

```
        while ((feof(trace) == 0) && (time < tracefilesize)) {
            time=time+1;
            if (fgets(LINE,MAXLINELENGTH,trace) == 0) break;

            if ((time % (REPORTTIME)) == 0)
                mexPrintf("\rtime: %12d     %2.0f percent
complete",time,(((float)time * 100) / tracefilesize));

            //break the line into the two relevant fields
            //skip, if the line doesn't start with an M, or
            //if the address is out of the range we care about
            TYPE = strtok(LINE," ");
            if (strncmp(TYPE,"M",1) != 0) continue;
            ADDRESS = strtol(strtok(NULL," "),NULL,16);
            if ((ADDRESS < MAINLOWERLIMIT) ||
               (ADDRESS >= (MAINLOWERLIMIT + MAINSIZE + SPADSIZE)))
                    continue;

            //now the tough part... we've got to correspond the
            //memory address with the object being accessed
            for(count = 0;count<numobjects;count++)
            {
                if ((ADDRESS >=
                    (int)mxGetScalar(mxGetField(OUTPUT,count,"START")))
                && (ADDRESS <
                    (int)mxGetScalar(mxGetField(OUTPUT,count,"END")))) {

                    temp = (time / DELTAT);
                    ACCESS[count][temp]++;
                    pointer=
mxGetData((mxGetField(OUTPUT,count,"total")));
                    (*pointer)++;
                    pointer=
mxGetData((mxGetField(OUTPUT,count,"mean")));
                    (*pointer)+= time;
                    count = numobjects;//stop the for loop, we've already
found it.
                } //end IF match
            }//end for (searching for right element
        }//end while (reading trace file)

        //now copy the ACCESS array
        mxAddField(OUTPUT,"ACCESS");

        for(count = 0; count < numobjects; count++) {
            pointer = mxCreateDoubleMatrix(1,numbins,mxREAL);
            pointer2 = mxGetPr(pointer);
            mxSetField(OUTPUT,count,"ACCESS",pointer);

            for(temp = 0; temp < ((int)numbins); temp++) {
                pointer2[temp] = ACCESS[count][temp];
            }
        } //end for loop (copying ACCESS)

        mexPrintf("Writing temporary output file\n");
        //write OBJECTS to a .mat file
```

```
    obj = matOpen(outfilename,"w");
    matPutVariable(obj,"OBJECTS",OUTPUT);
    matClose(obj);

    //end.. clean up.
    fclose(trace);
    mxFree(LINE);
    mxDestroyArray(OUTPUT);
    for(count = 0;count < numobjects;count++) mxFree(ACCESS[count]);
    mxFree(ACCESS);
}
```