Extended Architectural Enhancements for Minimizing Message Delivery Latency on Cache-Less Architectures (e.g., Cell BE)

by

Anthony Kroeker
B.Eng., University of Victoria, 2009

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

Extended Architectural Enhancements for Minimizing Message Delivery Latency on Cache-Less Architectures (e.g., Cell BE)

by

Anthony Kroeker
B.Eng., University of Victoria, 2009

Supervisory Committee

Dr. Nikitas Dimopoulos, Supervisor
(Department of Electrical and Computer Engineering)

Dr. Kin Li, Departmental Member
(Department of Electrical and Computer Engineering)

Dr. Sudhakar Ganti, Outside Member
(Department of Computer Science)

Supervisory Committee

Dr. Nikitas Dimopoulos, Supervisor
(Department of Electrical and Computer Engineering)

Dr. Kin Li, Departmental Member
(Department of Electrical and Computer Engineering)

Dr. Sudhakar Ganti, Outside Member
(Department of Computer Science)

ABSTRACT

This thesis proposes to reduce the latency of MPI receive operations on cacheless architectures, by removing the delay of copying messages when they are first received. This is achieved by copying the messages directly into buffers in the lowest level of the memory hierarchy (e.g., scratchpad memory). The previously proposed solution introduced an Indirection Cache which would map between the receive variables and the buffered message payload locations. This proved somewhat beneficial, but the lookup penalty of the Indirection Cache limited its effectiveness. Therefore this thesis proposes that a most recently used buffer (i.e., an Indirection Buffer) be placed in front of the Indirection Cache to eliminate this penalty and speed up access. The tests conducted demonstrated that this method was indeed effective and improved over the original method by at least an order of magnitude. Finally, examination of implementation feasibility showed that this could be implemented with a small Cache, and that even with access times 6x slower than initially assumed, the approach with the Indirection Buffer would still be effective.

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

I would like to thank:

My Supervisor, Dr. Nikitas Dimopoulos For his patience, mentoring, and support. It has been immeasurably helpful in my research, my studies, and my development as an Engineer.

My Fiance, Kelley Fea For putting up with my insane sleep schedule, and still managing to be endlessly loving and encouraging. :)

DEDICATION

To all the bunnies...who are no longer with us.
;(

Chapter 1

Introduction

1.1 Statement of Problem

This thesis' goal is to improve high performance computing by reducing the delay which is a result of the interprocess communication during computation. The area of focus is the message passing latency, and how it is possible to reduce it by removing additional message copying during receive operations. On architectures like the Cell BE[9] which are cacheless, and instead use a scratchpad memory, there is the potential for architectural changes (adding in a small Caching mechanism) which can achieve this latency reduction quite efficiently.

The exploration was conducted using parallel benchmark instrumentation and data collection, and then subsequent single-core simulation of the root (node 0) processor using this collected trace data. These simulations produced detailed variable access patterns and addresses which were used to calculate the performance impact of the proposed architectural changes. Finally, there is an initial investigation into the implementation of the new caching environment and how the proposed methods are effected by the timing, and how these methods can be used to optimize the energy usage of the cache.

1.2 Summary of Study

This thesis will first prove that the Indirection Caching mechanism (presented in full in section 2.7), previously introduced by Khunjush [12] [13], is generally more effective than the Classical message copying approach. (The Classical message copy-

ing approach involves copying the message from main memory to the lowest level of the memory hierarchy). This proof of effectiveness is achieved by providing robust results based on a wide range of: benchmarks, benchmark sizes, and timing assumptions. This testing is then extended to prove that a new Indirection Buffer mechanism (detailed in section 2.3.1), which can be added to the Indirection Cache mechanism, further improves its performance. Finally, it is shown that this can all be implemented with minimal energy penalty, if the right mechanisms are in place to avoid redundant Indirection Cache accesses.

The main idea behind the Indirection Cache is that it allows messages to be copied directly to buffers in the lowest level of the memory hierarchy (thereby avoiding at least one additional copy usually present in the classical approach). This works because the Indirection Cache exists to link these buffered locations to their final memory location. The extension of this idea is to reduce the Indirection Cache lookup penalty by buffering the most recently used address location, and using it predictively. This Indirection Buffer mechanism improves latency, over the original Indirection Cache mechanism. This is especially important for improving upon instances where the Indirection Cache is slower than the Classical message copying approach. Although, new estimates of worst-case Cache access performance show that Indirection Cache would barely break even, the Indirection Buffer implementation under these same Cache access conditions is able to maintain a 1 to 2 order of magnitude improvement over the Classical message copying approach.

1.3 Literature Review

The main problem this Thesis aims to address is message passing latency in parallel computing applications. When a message is passed between cores it is copied into an outgoing network buffer, sent over the network; and then copied again from the receiving network buffer before it reaches its final location (and can be accessed). The proposed approach aims to reduce latency by removing an extra copy operation, and to do this by moving received messages directly into the lowest level of the memory hierarchy (eg. cache, scratchpad memory) as soon as they come in from the network. The approach used previously by Afshahi [1] and Khunjush [13] [12], and that this Thesis directly expands on, is introduced in section 1.3.5, as a lead up to the full discussion in the background Chapter 2. In general, the approach of manually manipulating the cache in this manner is referred to as Cache Injection [15].

This is just one method for potentially reducing latency, and has been explored by other reseachers. The following sections 1.3.1, 1.3.2, and 1.3.3 are current examples in this research area that relate to supercomputing applications. Another approach to solve message passing latency issues involves changing the messaging protocol for how messages are moved between nodes. Section 1.3.4 discusses this issue and explores how to improve the standard send and receive operations.

1.3.1 Cache Injection for Parallel Applications

The paper "Cache Injection for Parallel Applications" [14] looks at reducing the effects of the memory wall by using Cache Injection (moving messages directly from the network into cache). The general idea is to reduce latency for data accesses by employing one of three cache injection policies, caching the: message headers, message payloads, or both. Though their approach does reduce latency, their research focus was on increasing bandwidth, because the injection relieves pressure on the memory and because of the benchmarks they tested with. They conclude that cache injection effectiveness depends on: The choice between these 3 injection policies (with some benchmarks benefitting more from one than the other), the communications characteristics of the benchmark being tested, the target cache, and the severity of the memory wall. Their research confirms our approach, which is to test on a wide range of benchmarks because the caching effectiveness depends so heavily on the specifics of the application communication and data access patterns. Our research takes the approach of caching both the header and payload information, because the coherency of our mechanism relies on all this information being present. Also, our approach includes a separate cache for the messages, though their research does indicate that cache pollution would be negligible if sharing a cache was the only option.

1.3.2 Direct Cache Access for High Bandwidth Network I/O

The paper "Direct Cache Access for High Bandwidth Network I/O" [11] takes a system level view of the cache-memory-processor interaction, and what can be done to optimize it. More specifically, they are motivated by the throughput demands of 10Gb/s networks and how this can demand processor response times as low as 67ns. So, in this case extreme throughput demands go hand-in-hand with latency demands, especially when the data is coming from the processor and memory (ie. not just DMA'd directly out of memory). Their work looks quite extensively at

the coherency protocols that must be in place to keep the memory, proccesor, and network card synchronized via the chipset that ties them all together. The overall idea is that messages move directly into the processor-cache from the network card and the memory is kept in sync with these operations (and any evictions and cache changes that are propogated to memory as a byproduct). Compared to our approach their solution uses the existing caching structure as part of the entire system, just with new protocols. Our approach assumes explicit instructions to load and store to a specialized network cache, which adds to the existing system hardware. They touch on this aspect, because their recommendation is to only transfer directly to the highest level cache except when the cache is not used for other things. In this way our system allows the network cache to be lower level (and therefore lower latency) because of its separation from the main processor cache system.

1.3.3 Cache-Based Memory Copy Hardware Accelerator

The paper "Cache-Based Memory Copy Hardware Accelerator for Multicore Systems" [6] looks at removing excess (and latency costly) copying from the send-receive path of message transfers. They look specifically at multicore systems in a shared memory environment and how usually the message has to be copied multiple times: into a send buffer, into a shared location, and then into a receive buffer (with each of these pulling the necessary memory locations into cache). Their solution is to add a index table into the shared cache (eg. L2) of the multicore procesor, which can be used to update pointers to the messages as they move across the system (instead of having to copy any data). This is quite exciting, because it is essentially the shared memory equivalent of our Indirection Cache approach that we proposed for cacheless architectures (like the Cell Processor). Their system also suffers from a lookup penalty when accessing their index table, though they discount this cost as negligible (understandable considering the vast improvements their method offers over the traditional message copying via memory). They have essentially taken the problem and pushed it down a layer in the memory hierarchy, and removed some message copying. However, the individual cores are still stuck copying data in and out of their L1 caches (to/from the L2 cache). This is partly an artifact of the shared memory paradigm the cores are operating in, but also shows why examining other architectures like the Cell Processor is important. Our research also looks to drastically reduce the 'index table' (ie. our Indirection Cache) lookup penalty with the introduction of an Indirection Buffer.

1.3.4 Remote Direct Memory Access (RDMA) MPI

The paper "High Performance RDMA-Based MPI Implementation over InfiniBand" [16] looks at the Infiniband Interconnect and its RDMA ability, and how this can be used to accelerate message passing operations. They look at replacing the standard message passing send and receive operations with a RDMA write command. The problem that they encounter is that though RDMA is faster, it also must know the receive variable address ahead of time. This means that adapting it to transparently replace the standard MPI send/receive implementation requires additional control messages to be sent (which impact the latency). They get around some of these limitations by putting persistent buffers associations between processors, which last for the duration of the program, but this still has some limitations. The conclusion they reach is that for small messages it makes sense to use RDMA, but to still use the regular send/receive operations for the larger messages. Because of these limitations and because the RDMA is more network-centric than our approach, the two approaches are complimentary. In other words, RDMA is something we can use for actually getting the message across the network, and could be utilized by our mechanism which effects the first and last legs of the messages journey. Our proposed approach on the Cell processor already has this as part of its assumption as the underlying message transfers between cores can use DMA, put, or get operations. The key is that our proposed changes mean that messages can be sent without knowing the final destination address, and still be zero-copy, because of the Indirection Cache.

1.3.5 Khunjush's Work

Farshad Khunjush did initial work on zero-copy message transfer through the use of a specialized cache, which he detailed in his Thesis [13]. This work introduced the idea that of direct to cache transfer of messages, and the ability to do this even when the final destination address was unknown (late binding). He then followed up on this work by examining the Cell processor, and how this approach could work on cacheless architectures [12]. As mentioned earlier, the main idea is to have an Indirection Cache which can be used to track the messages stored in the local scratchpad memory (and link them to their actual destination addresses). This Thesis expands directly on this idea by conducting further testing with more benchmarks, different sizes of benchmarks, and under different cache timing assumptions. This Thesis also looks to reduce the latency further by introducing a new concept: an Indirection

Buffer. Buffering the most recently used Indirection Cache line, and using this value predictively (so the processor can fetch the data immediately) will almost eliminate the Indirection Cache lookup penalty. The following background chapter starts with what is essentially an extended Literary Review of Khunjush's work, because it is necessary to understand in depth how his mechanisms work, before proceeding to this Thesis's contributions.

1.4 Outline

The following will be covered in this thesis.

Chapter 1 Introduced the main claims that this Thesis makes and examined related work.

Chapter 2 Explores the problem to be solved, and specifically: Brings the reader up to speed on the past research which this Thesis builds upon, and then explains the theory of the current approach.

Chapter 3 Gives the research methodology used to obtain the results and conduct the analysis.

Chapter 4 Shows the main results in detail through tables, and summarizes them with graphs. Time is taken after the presentation of each portion of the data to analyze and highlight the key aspects of the results.

Chapter 5 Drills down into the specifics of the Caching implementation and limitations, and further justifies the need for the Indirection Buffer.

Chapter 6 Summarizes the main points made in the analysis, and rehighlights the benefits of the new method. It finishes by discussing several areas of future work.

Chapter 2

The Problem to be Solved

2.1 Goal

The overarching goal is to reduce message passing latency, in MPI environments. The proposed approach minimizes the latency introduced when messages are accessed for the first time, and are copied from the memory/network buffers into cache. The original idea [13] was to architecturally add caches to the processor which would be populated upon message arrival with: Message data, and the necessary meta-information to facilitate direct access from MPI functions requesting these messages. The new approach is to leverage the scratchpad memory on processors (eg. The Cell, see section 2.2.3) to store the payload portion of the message and only store meta-information in a small cache. This allows for the implementation of a smaller cache large enough to hold the payload memory addresses, and to hold the MPI meta-information. Initial research has shown that this new caching structure improves on the access latency, but introduces a small per-access penalty as an artifact of the cache design. The current goal is to greatly reduce this penalty by adding additional buffers to the architecture, and to test the design with a wider range of benchmarks. These buffers hold the most recently used cache line, to exploit temporal locality, and save time by fetching data in parallel with the cache lookup. This is the focus of the latest research efforts which led to the results and analysis in this thesis.

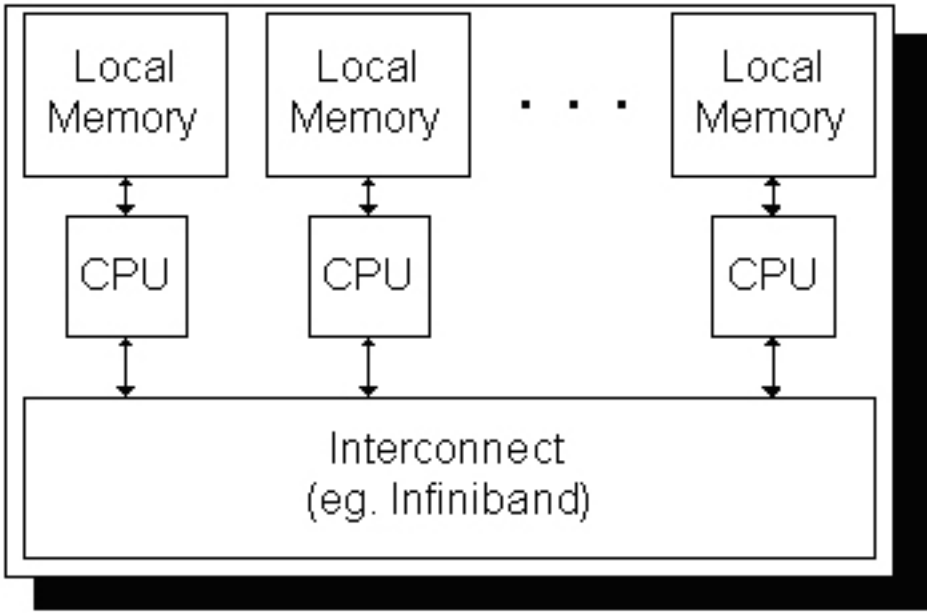

Figure 2.1: Distributed CPU & Memory Architecture [2], [8]

2.2 Background

The initial investigation and simulation of this technique was done by Farshad Khunjush, in his thesis "Architectural Enhancement for Message Passing Interconnects" [13], and subsequent paper "Architectural Enhancement for Minimizing Message Delivery Latency on Cache-Less Architectures (e.g., Cell BE)" [12]. The following sections will cover the background material needed to fully understand the methods used, and the results achieved.

2.2.1 Problem Space

To construct large computer programs that simultaneously run on multiple computer processing cores requires that information be exchanged between these cores during computation. One approach is to explicitly send and receive messages between cores.

"The Message Passing Interface (MPI) has emerged as the quasi-standard for message passing libraries" [10, p.681]. The interface provides many functions which expedite information exchange and ease of programming [21] [18]. For this research however, the two base functions "MPI_Send" and "MPI_Recv" (receive) are chosen to demonstrate the architectural enhancements (their declarations are summarized in Table 2.1). These functions were chosen because they are commonly used across

MPI benchmarks, and are the underlying mechanism leveraged for more complex functions. The way these functions are paired, requires that for every send call there is a matching receive call. In this way, every message has a source and destination. Their operation is symmetric as well, with sent messages being created, buffered, and sent over the network; while incoming messages are received from the network, buffered, and consumed. This is illustrated in Figure 2.2.

int MPI_Send		int MPI_Recv
Output Parameters		Output Parameters
NA	initial address of receive buffer (choice)	buf
NA	status object (Status)	status
Input Parameters		Input Parameters
buf	initial address of send buffer (choice)	NA
count	# of elements in buffer (nonneg integer)	count
datatype	datatype of each buffer element (handle)	datatype
dest	rank of destination (integer)	NA
NA	rank of source (integer)	source
tag	message tag (integer)	tag
comm	communicator (handle)	comm

Table 2.1: MPI_Send and MPI_Recv Function Definitions [20] [19]

It is important to note that the MPI_Recv function is essentially a blocking call, dependent on getting information from another computing core (unlike the send operation which only has to wait until the data has left over the network). Past a certain point, the received data is critical and the computation will not be able to proceed without it. Therefore, the latency of the receive operation is of great importance, and is one reason that the MPI_Recv side of the exchange has been the focus of this research. Another reason is that the dependency on the received data provides the opportunity to exploit temporal locality when dealing with the MPI_Recv latency problem.

Focusing more specifically on the MPI_Recv function, it is important to understand the two scenarios that arise during its use. The first case is that the MPI_Recv function is called at the destination core, before its corresponding message has come in over the network from the source core. In this case the message is considered 'late', because the destination core must wait for it to arrive. The system has the opportunity to pre-allocate space for the expected message, and record the necessary meta-information to facilitate the message receive operation. Then, when the message arrives over the network it is transferred into the appropriate memory location

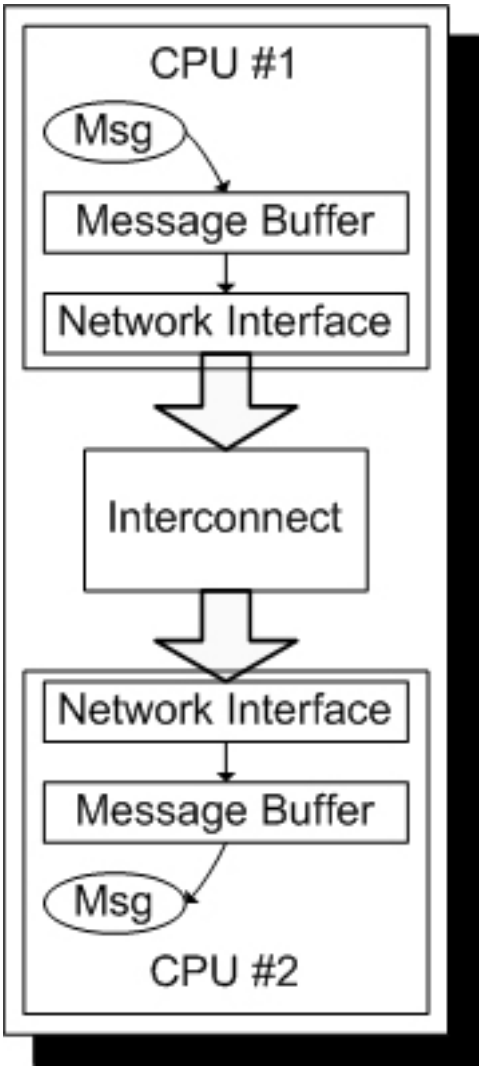

Figure 2.2: MPI_Send and MPI_Recv Communication Diagram

(corresponding to the receive variable address which was specified as part of the MPI_Recv function call).

The second case is when the MPI_Recv function is called at the destination core, after the corresponding message has arrived over the network. In this case the message is considered 'early', because the destination core is not yet ready to receive the message. If this is the case, the message will be located in a temporary network buffer, along with the identifying meta-information sent along with the message. In particular, it will be missing the receive variable address, which denotes where the data will be copied for use. When the MPI_Recv function is called, the core sees that the message has arrived early and copies the message from the temporary network buffer to its final destination at the receive variable address. These two scenarios must be understood to realize the potential for improvement, specifically in the second scenario where there is an extra copying operation that occurs.

The alternative MPI functions that are commonly used in place of the blocking calls (MPI_Send & MPI_Recv) are the non-blocking functions MPI_ISend and MPI_IRecv. These two functions return immediately, and a call to MPI_Wait can be used to determine if they have completed. This allows the programmer to interleave communication and computation, which helps to reduce the time wasted sitting idle waiting for messages. This is ideal because it allows the computation grain to be reduced further, as the time penalty for communication is reduced, which allows for more parallelization. However, even if the interleaving is optimal, the first time a message is accessed a penalty is incured beause the message must be copied from memory, into cache, before it can be used. Therefore the granularity's lower limit is this copying delay, and if it too can be reduced or eliminated then the achievable granularity of the computation will be reduced as well. The immediate movement of the received message into the lowest level of the memory heirarchy (eg. cache or scratchpad memory) does just that. Therefore, independent of programming approach (ie. nonblocking vs blocking MPI functions) the proposed caching method sets a new lower limit for the computation grain.

2.2.2 General Network Cache Approach

The approach presented by Khunjush in his thesis was to "Achieve zero-copy communication in message passing environments" ([13, p.31]). The idea was to leverage the predictability of message consumption patterns to bring the message payloads into

the lowest level of the memory hierarchy, the cache. This would make the messages consumable by the receiving process without additional delay. The proposal was to create a cache to hold each piece of the message's identifying meta-information (network tag, message id, and process tag), as well as the message payload. In the ideal situation, the cache was considered to be fully associative, and this also corresponds to the high-level conceptual case illustrated in Figure 2.3. However, having a single cache hold all this information (and still be associatively searchable) could not be feasibly implemented. Therefore, Khunjush proposed using several smaller set associative caches (one for each searchable field). The cache fields were linked together so that searching by any one piece information would allow the data line in the message paylod cache to be resolved.

This new multi-cache approach introduces an indirection step, where the field search returns the location in the message payload cache, and then the payload data is pulled from that cache. The trade-off then becomes whether to: Access the payload data via the cache lookup everytime (and incur the indirection lookup penalty), or to pay the lookup penalty once and then copy the payload data into the main data cache (thereby avoiding subsequent lookup penalties). These two approaches were studied by Khunjush, and it was determined that the indirection penalty was generally better than the data copying approach. This indirection based message caching architecture's operation is described in detail in the following paragraphs.

The operation of the cache was split in to two scenarios: Late binding, where the MPI_Recv call comes in after the message has arrived, and Early binding where the MPI_Recv call comes in before the message has arrived. The more complex case is the Late binding. When early messages arrive the payload is copied into the cache immediately, and the network tag is updated to point to the network buffer address (that the message was received on). The message id field is also updated, using the information carried in the MPI message envelope. Then, when the MPI_Recv call is made, the correct cache line is found through a lookup of the message id. The process tag field is filled in with the receiving variable address, and the message id and network tag fields are nullified. The network buffer is also freed because the message data exists in the cache, and upon eviction will be moved to the receive variable location. (The full eviction and replacement behaviour is detailed in Khunjush's thesis[13]). Going forward, only the process tag field is required to lookup the data location in the cache. These steps are outlined in Figures 2.4 and 2.5.

For Early binding the MPI_Recv call comes in first and a cache line is reserved

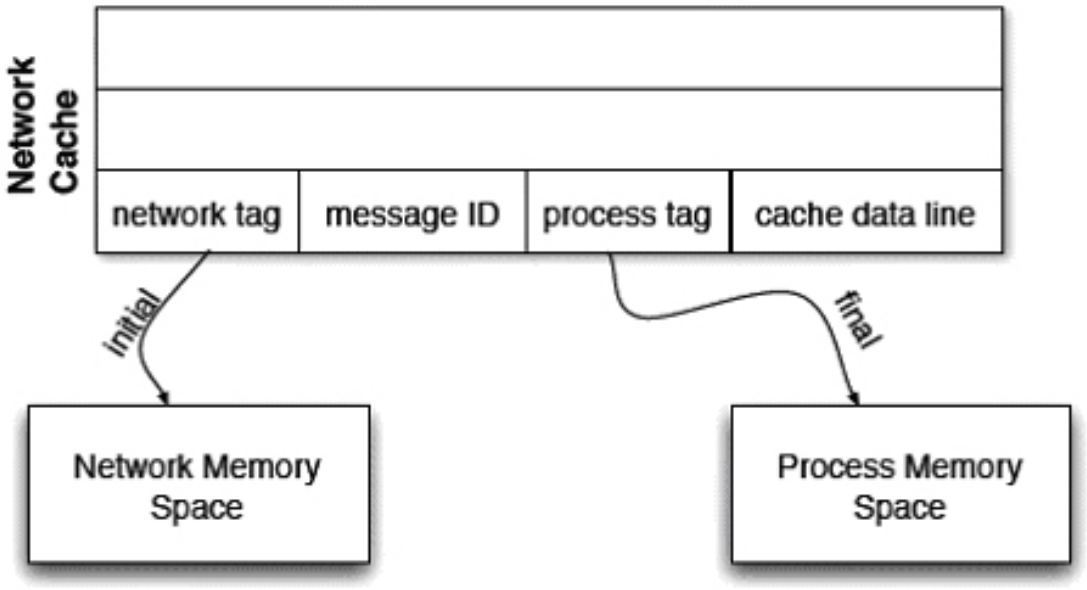

Figure 2.3: Network Cache Architecture - Khunjush's Thesis

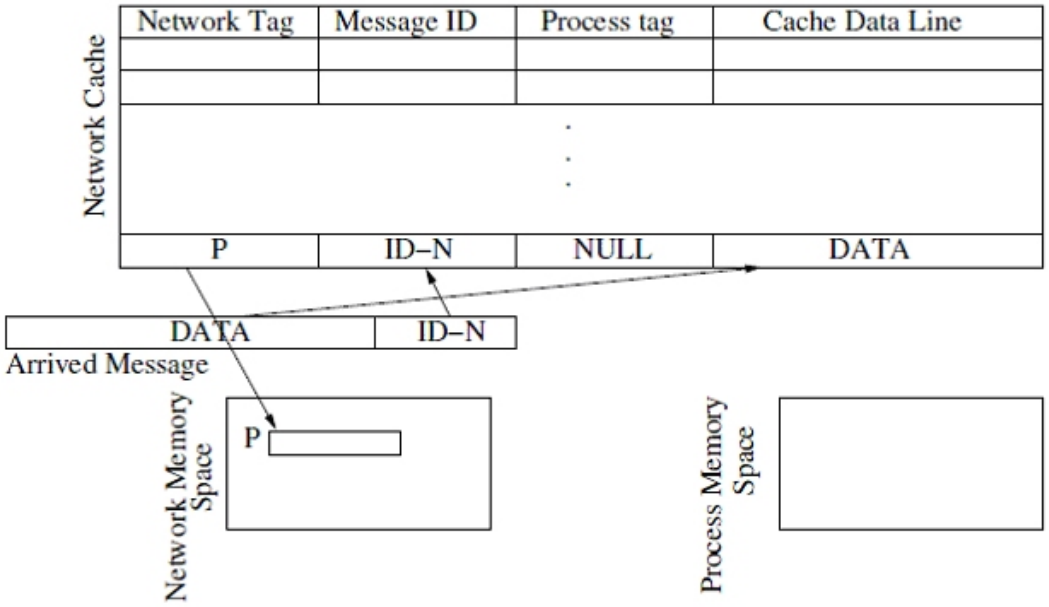

Figure 2.4: Network Cache after Message Arrival, but before Late Binding

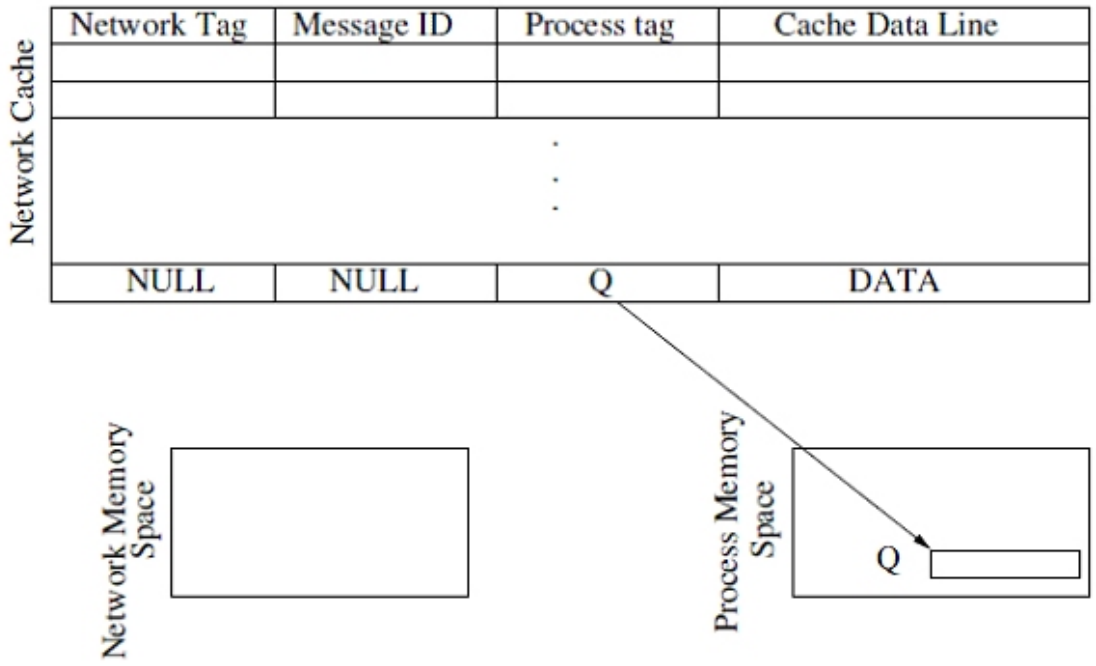

Figure 2.5: Network Cache after Late Binding

for the payload. At the same time the message id is filled in with the MPI identifying information from the function, and the process tag is filled in with the receiving variable address. Then, when the message arrives over the network it is immediately moved into the cache, and the network tag remains null. Going forward only the process tag field is required to lookup the data location in the cache, just like the Late binding scenario.

Recall that the classical data transfer time is defined as the amount of time required to copy the message from its main memory network buffer to its final receiving variable location. Khunjush's work proved that the proposed caching concept reduced the message access latency introduced by this classical transfer time. However, it also shows that there is room for improvement because of the overhead associated with each message cache access (because of the lookup delay to retrieve it from the network cache), and the complexity of the multiple-cache mechanism. Additionally, limits to the cache and cacheline size put an upperbound on the size of the message payloads that they could accomodate. Therefore, the approach lends itself to the idea of leveraging cacheless architectures which use a large scratchpad memory as their lowest level of the memory heirarchy. This approach was explored in a subsequent paper [12] by Khunjush, and is discussed in the following sections.

2.2.3 The Cell Processor

The Cell Processor, also known as the Cell Broadband Engine Architecture (CBEA), is a heterogeneous multicore processor from IBM [9]. It was originally created as a collaboration between Sony, Toshiba, and IBM in 2001. Its purpose was to meet the needs of the PlayStation 3 game console, but because of its abilities, it has proven to be quite well suited for super computing applications (eg. the Roadrunner supercomputer [17]). These extreme number crunching abilities derive from its unique heterogeneous multi-core architecture. The Cell is heterogeneous because it is made up of a main PowerPC core, and several (usually 8) smaller co-processor cores. The PowerPC core, or PPE, is fully featured with support for the Power ISA. Each of the co-processors, or SPEs, is a 128-bit RISC processor, and is connected to the PPE and other SPEs via a ring bus. This interface is done using a dedicated memory controller, which allows the SPEs and PPE to initiate local or remote DMA transfers between cores. Additionally, each SPE has a 256KB local scratchpad memory, which can essentially operate as a form of manually controllable cache. (i.e. it is not

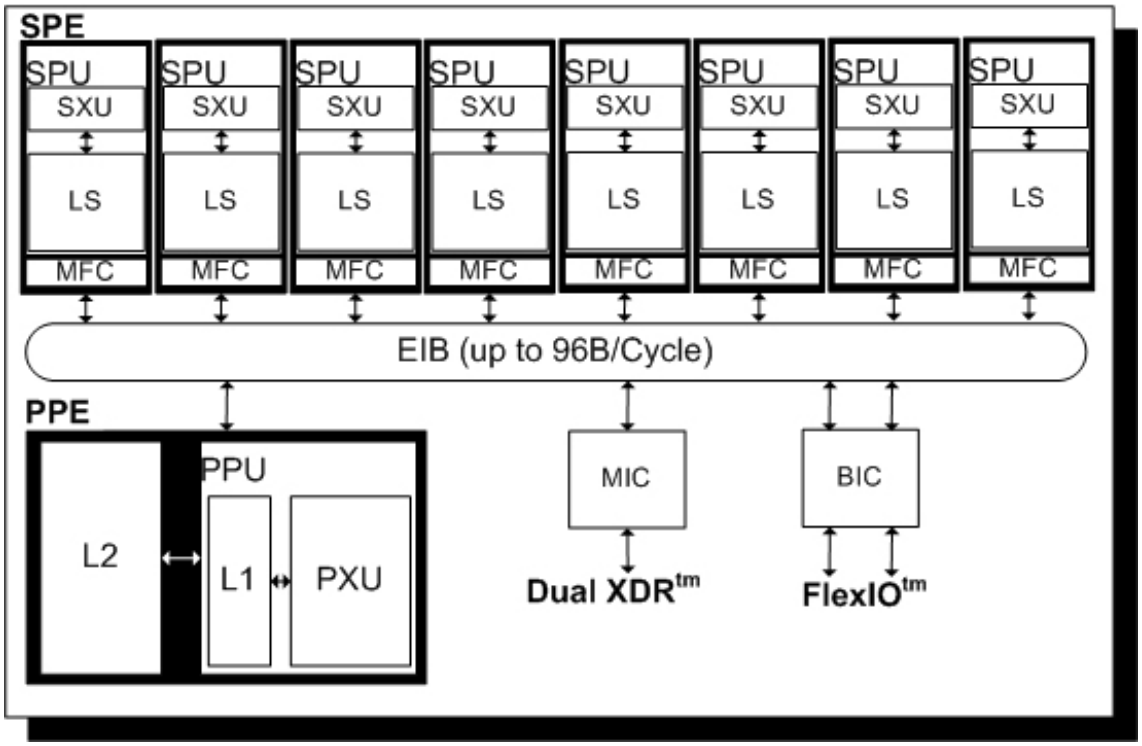

Figure 2.6: The Cell Processor ([9, p.236])

transparent during memory accesses, and must be addressed/controlled implicitly by its SPE). The Cell processor is shown in Figure 2.6.

2.2.4 Cacheless Architecture Approach

Khunjush's paper [12] expands on his thesis, uses the knowledge gained from the network cache investigation, and applies it to the Cell processor. In this regard, the performance measurements and testing from the thesis can also be used to help gauge the performance of the new methods proposed in this paper. To work on the Cell, the caching mechanism was modified to work with the local scratchpad memory available to each SPE. Therefore, it was proposed that the payloads would be stored in this local memory, and only a small Indirection Cache would be added to the architecture. The basic idea being that the Indirection Cache would only track the scratchpad memory address of each message, rather than storing the entire payload, thereby reducing the size and allowing the cache to work for all message sizes.

The structure of this Indirection Cache is almost the same as the one presented previously and is comprised of 7 fields, illustrated in Figure 2.7. For each message

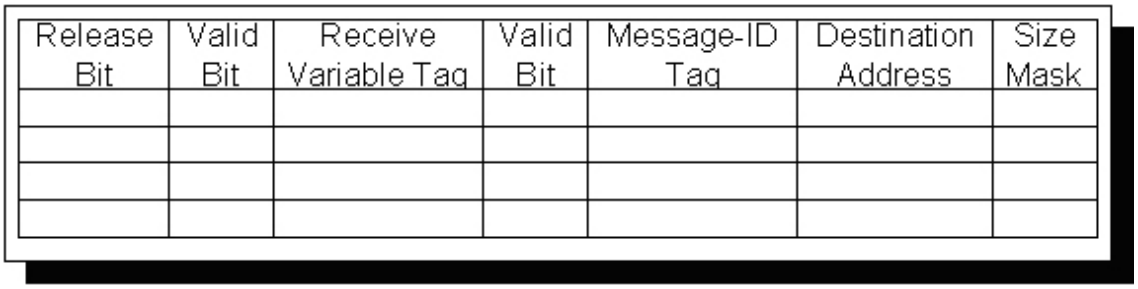

Figure 2.7: Indirection Cache - Khunjush's Paper

it tracks the receiving variable tag, message id tag, destination address, size mask, release bit, and valid bit. The last three fields support the central operation of the cache. The release and valid bits are used for managing the lifetime of the message within the cache, and can be leveraged by the garbage collection and eviction mechanisms. The release bit is used when the entire line needs to be invalidated and freed up. For example, the release bit will mark the entire line as expired because of a subsequent replace and therefore the line can then be reused for tracking new message arrivals. The first valid bit is paired with the receive variable field. It is only marked valid when it has been populated with a valid receive location. The second valid bit is paired with the message id tag (and is marked valid when the line is associated with a valid message id). These valid bits help track the current state of the receive operation. For example, when a message first arrives the message id tag will be valid, but then once the message has been bound to its receive variable location the id is marked invalid. The size mask is simply in place to track the current message's payload size, and used for required boundary checks and buffer allocations/deallocations.

This leaves the three main tags used for the indirection: receiving variable tag, message id tag, and destination address tag. The receiving variable tag holds the address of the 'receive buffer' used in the MPI_Recv call. The message id tag holds the unique portions of the MPI meta-information that identify the message payload. The destination address tag contains the payload buffer address (in other words, the actual point in local memory where the payload is located). With this structure, the indirection is managed and setup in much the same way as the cache outlined in the previous section.

The local memory where the message payloads are stored is designed to hold the only local copy the program will require. Therefore, a section of the local scratchpad

memory is reserved for the creation of message payload buffers. Incoming messages are copied into these buffers, regardless of the receiving variable address provided with the MPI_Recv call. This works because the Indirection Cache allows the mapping between a receiving variables address and the actual payload buffer address. This mapping requires that the Indirection Cache be used each time a message payload is accessed, which means that a lookup must occur. This lookup takes time and therefore the program incurs an indirection penalty (which is at least one cycle, but dependent on the Indirection Cache implementation). This is problematic, and hiding (or at least reducing) this penalty is the focus of this Thesis.

A key part of the cache operation is how it can actually be used by MPI programs. Some of the cache operations are triggered automatically (eg. when a message arrives), but other operations must be done more explicitly. To load and store message payload data (ie. the data is in a receive variable associated with an MPI receive call) then two new instructions must be used: load* and store*. These new instructions indicate that the access should occur to memory, via the Indirection Cache. So instead of going to the memory address of the receiving variable, this address is used to search the Indirection Cache for the network buffer address (where the data actually resides). Once the search has returned the associated network buffer address the data will automatically be loaded or stored, to or from this location.

The operation of the Indirection Cache proposed for the Cell follows the same principle of dealing with the following two scenarios: Late message arrival, and early message arrival. The first case is simpler to handle because the MPI_Recv function call precedes the message arrival. This means that the cache fields that hold the receiving variable tag, message id tag, destination address, and size mask can all be populated ahead of time. The destination address will contain the location of the payload buffer preallocated for this message. Then, when the message arrives the system only has to move the payload into this location and fill out the rest of the cacheline (and unset the message id tag valid bit). For the early message arrival, the Indirection Cache fills out the fields for message id tag, destination address, and size mask. The data is then moved into a just-allocated payload buffer that the destination address points to. Then, when the MPI_Recv call is made the receiving variable tag and valid bit are set. Figures 2.8 and 2.10 outline these two scenarios, and Figure 2.9 emphasizes that resolving the destination address requires searching (which incurs an indirection penalty).

After both these cases, when the cache fields for the message are fully populated

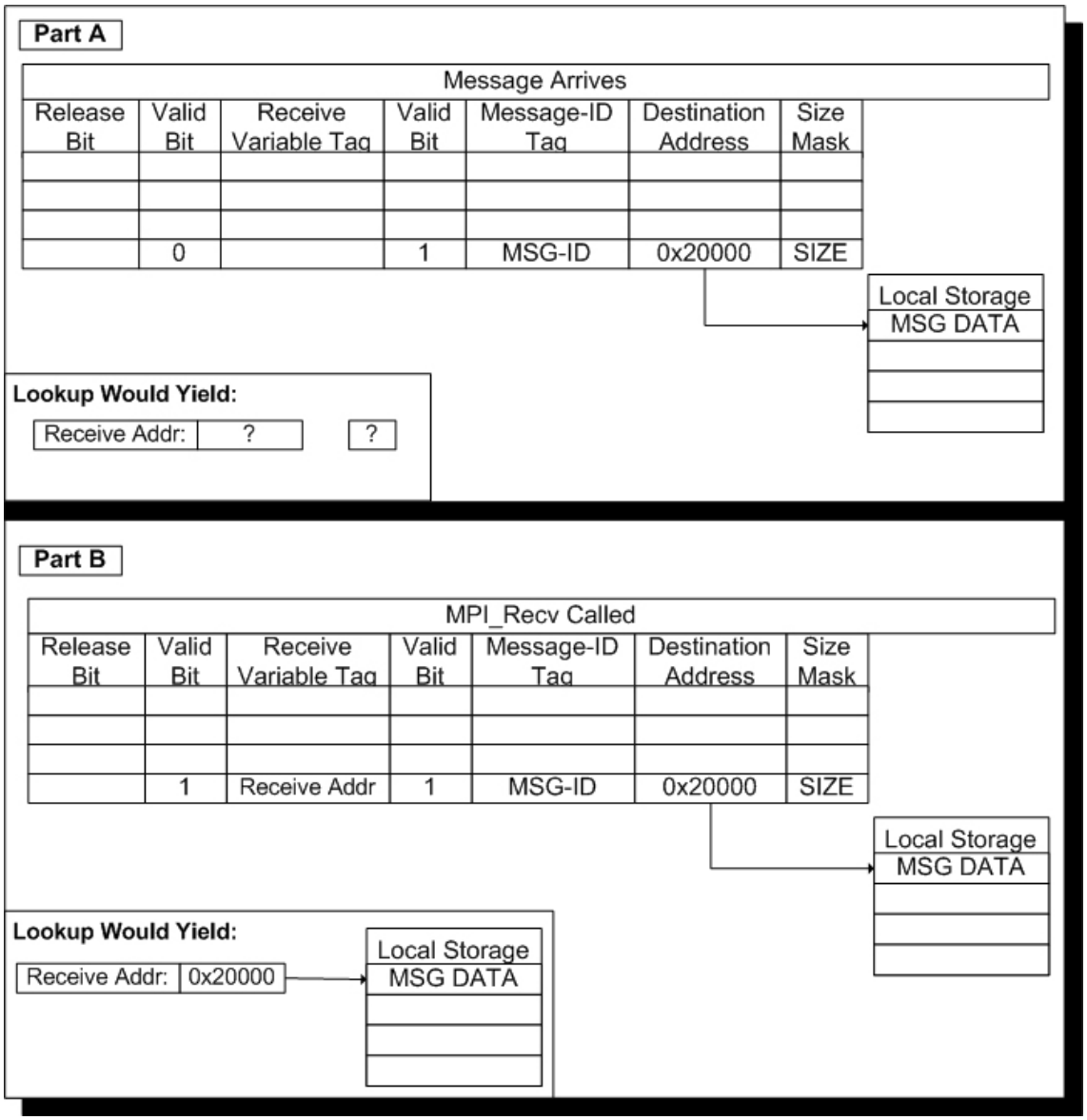

Figure 2.8: Early Message Arrival (Late Binding)

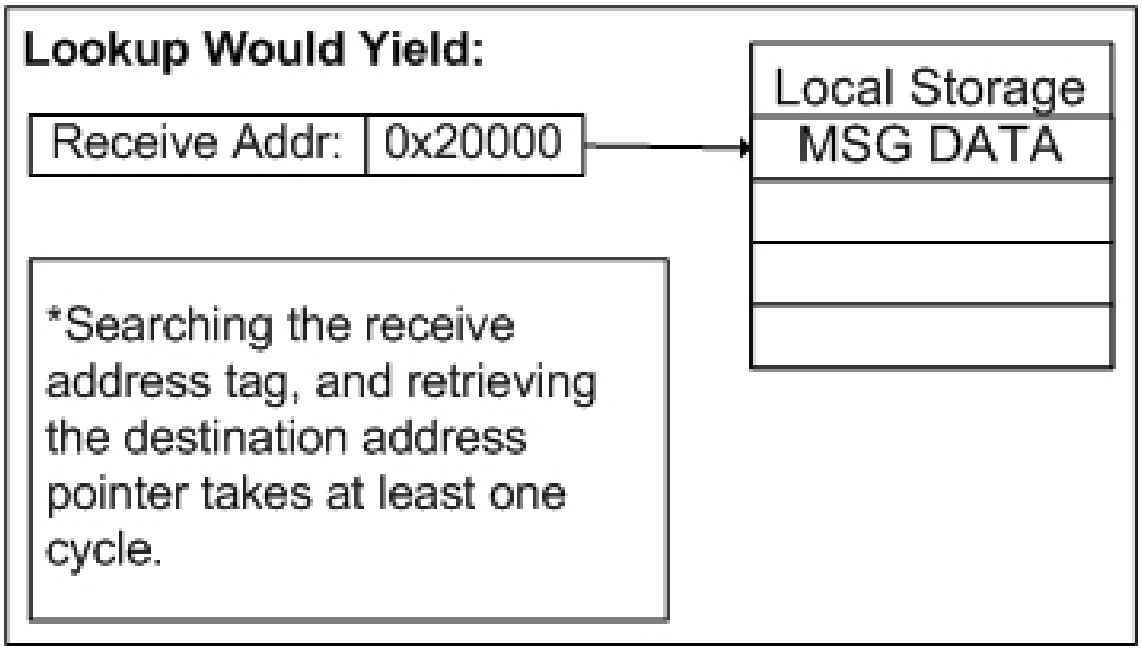

Figure 2.9: Indirection Cache Access Penalty

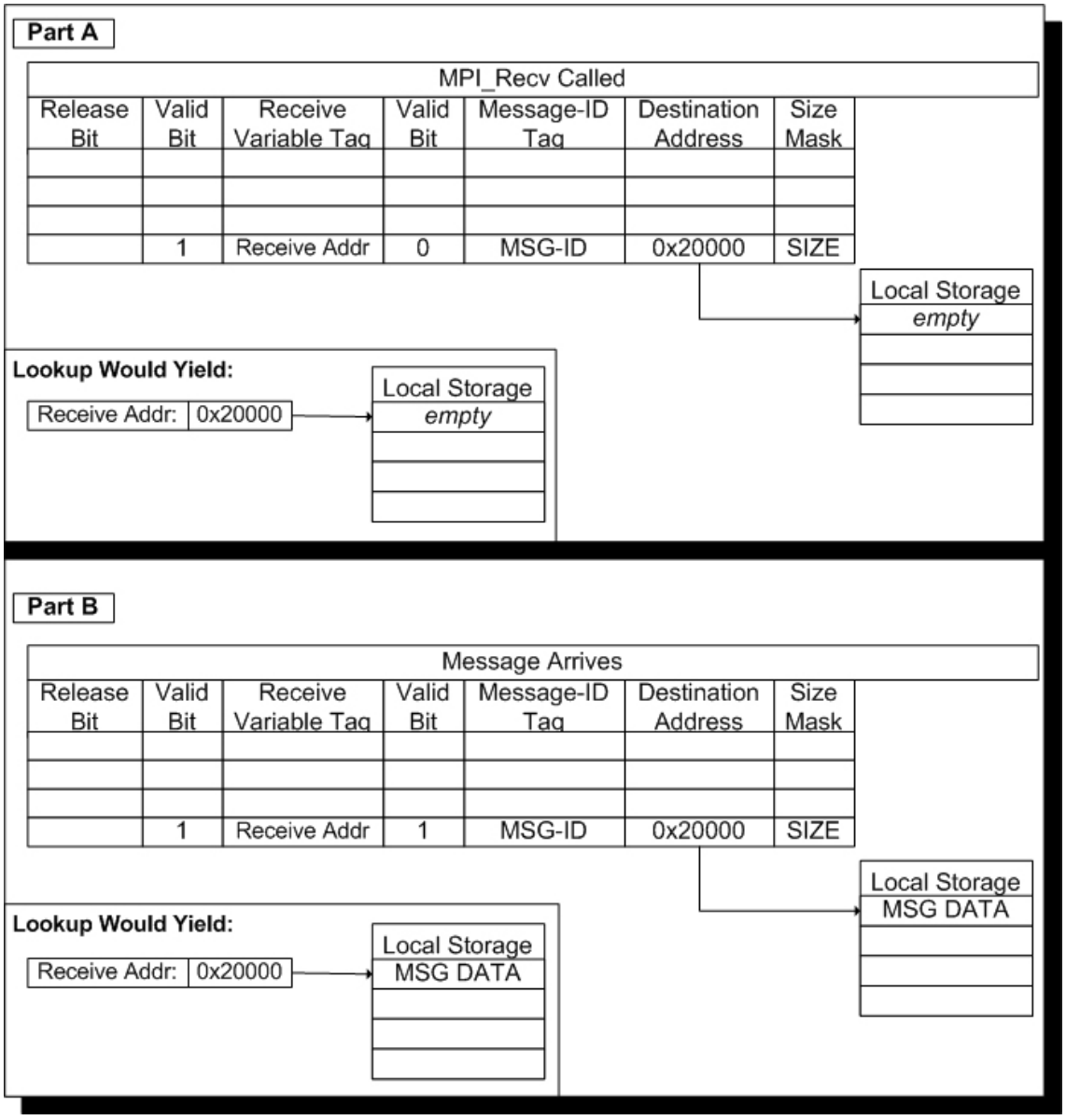

Figure 2.10: Late Message Arrival (Early Binding)

(i.e. Valid bit is set), then all subsequent requests for the receiving variable use the Indirection Cache. This means that load*/store* instructions tell the processor to: Go to the Indirection Cache, search for the receive variable address associated with the instruction and its corresponding destination buffer address, wait for this search to complete (the indirection penalty), then using the retrieved destination buffer address go to memory and load*/store* the data.

For this Thesis, it was initially assumed that a fully associative cache would be used for the Indirection Cache, but then steps were taken to investigate how well this design could generalize to other caches. Though fully associative is potentially impractical for implementation it allowed for a quick initial evaluation, in part because of the assumption that the replacements would be minimal. Also, for this initial analysis it was assumed that it would take one cycle to determine the actual payload location when searching the receiving variable tag and then retrieving the destination address (ie. The indirection penalty). The theory, also applied in Khunjush's Thesis [13], is that the cycles lost to this indirection, cost less than copying the message from a temporary network buffer to the final receiving variable location.

The Paper [12] concludes that this method does show potential, even with the indirection penalty. It also concludes that further testing would be warranted with more benchmarks, to see if the results hold over a wider range of data. The new approach presented in the next section looks to address the indirection penalty issue, and the results reported later on in this document address the second. They also go a step further to investigate these issues, by reanalyzing the results with different Indirection Cache architecture assumptions.

Figure 2.11: Indirection Buffer Structure

2.3 Current Work

The research presented in this thesis continues and expands the research Khunjush introduced, by further exploring the applicability of these techniques in cacheless systems. In addition to expanding on the architecture (detailed in section 2.3.1), the new research explores new optimizations in the experimental methodology (detailed in section 3 and Appendices), and drills down into the Indirection Cache design parameters (detailed in section 5).

2.3.1 Buffered Cache Approach

The approach chosen to overcome the cache indirection penalty was to add in an Indirection Buffer which holds the address of the last destination address accessed. This means that the last destination address is known immediately, without needing to search and access the Indirection Cache. This is illustrated in Figure 2.11.

This destination address is used immediately when an access request comes in. Meanwhile, in parallel, a check is done through the traditional Indirection Cache access method. If the address matches, then time has been saved and nothing needs to be corrected. If the buffered address was incorrect, then there is an instruction/pipeline rollback, and the calculation proceeds with the correctly fetched data. The rollback mechanism assumes that the processor pipeline can buffer instruction commits until the address is verified, so that the register or memory state is not erroneously updated by an incorrect load*/store* operation. This approach assumes that the underlying processor architecture is out-of-order, and these instruction buffers (and rollback mechanism) are already supported. Therefore, in the worst case the cache access penalty is the same (ie. the rollback has the same effect as just pausing the pipeline and waiting for the address to resolve), but whenever the buffer contains the correct address, time is saved. This approach exploits the principle of temporal locality, and if a low miss-rate of the Indirection Buffer is achievable, would provide significant time savings over the initial Indirection Cache structure.

The Indirection Buffer is only marked invalid when there has been a cache eviction, or the current receive variable is being replaced with a new receive call in the Indirection Cache (used for this Thesis). An alternative implementation would be to remove the validity check, and simply let these accesses fail like any other misprediction. However, including the valid bit removes wasteful extra memory accesses, for definite mismatch cases (ie. when the buffer is marked invalid). The general operation is illustrated in Figure 2.12.

An extension of this method is to use multiple Indirection Buffers. In the case of two Indirection Buffers, the least significant bits of the receiving variable tag would be used to partition all the variables into two sets. Ideally this would increase the performance of the cache in circumstances with heavy access interleaving between two receive variables. This approach, if successful, could be extended to N-Indirection Buffers, thereby partitioning the accesses into N sets. The 2 buffer case is illustrated in Figure 2.13

2.3.2 Energy Saving Option

The Indirection buffer, as currently described, will use the destination address while checking the Indirection Cache in parallel. This means every time the cache must be searched and accessed, which costs energy. If the cache only had to be activated for cases where the Indirection Buffer was wrong, then energy could be saved. In fact, the energy savings should be proportional to the percentage of correct predictions (i.e. the more accurate the Indirection Buffer is, the less energy is used accessing the cache). The question is: How to know if the Indirection Buffer contains the correct value? Some of the cases where it is wrong are covered by the valid bit (eg. if a new message was received at the currently buffered receive variable), but to always know when it is correct requires more information to be stored in the Indirection Buffer. The simplest mechanism is to store the receive variable tag alongside the destination address, and to compare this value to the receive variable passed in for the access. The size cost of this mechanism would be at most 64 bits to store the address, and less if there are N-indirection buffers (ie. only the higher order bits would need to be stored). The time cost would be negligible, as it is assumed that the bit comparison could be achieved asynchronously with combinatorial logic and then would either: Use the destination address from the buffer, or activate the Indirection Cache access.

This approach would eliminate the need for an out-of-order assumption which

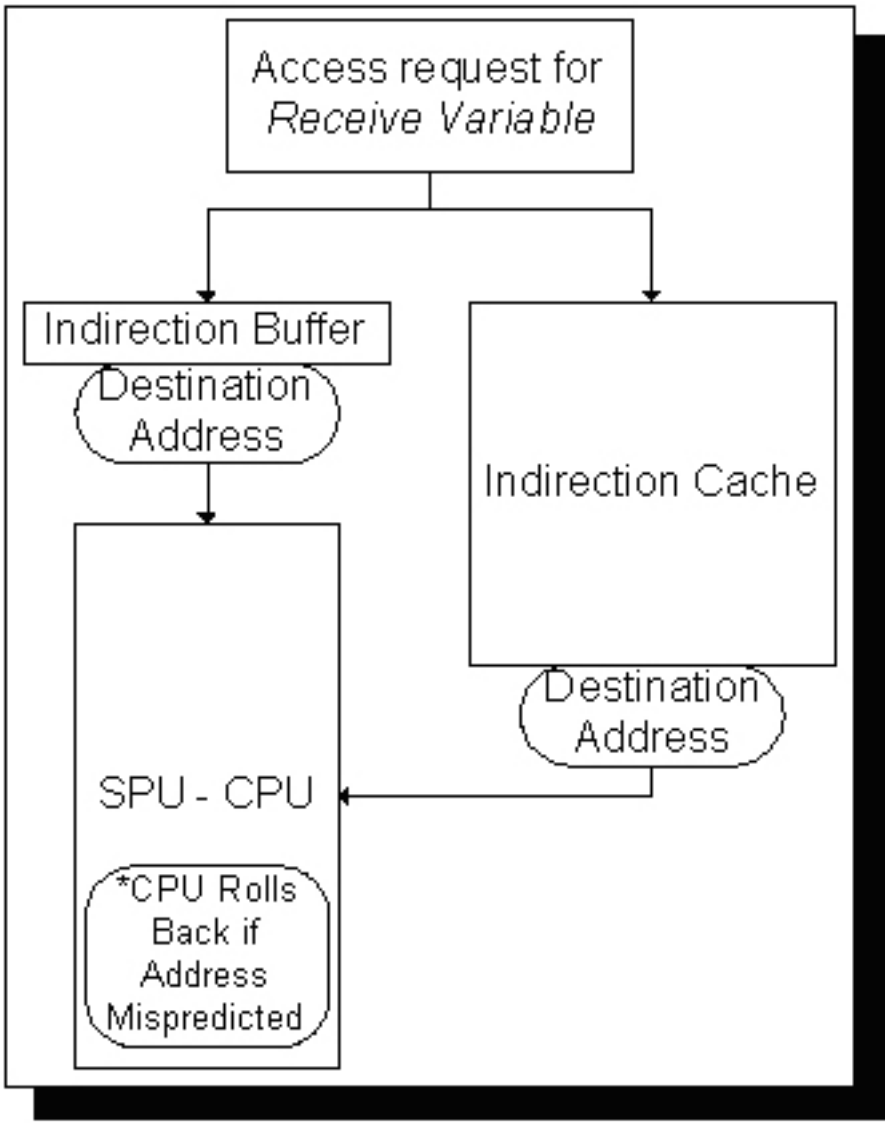

Figure 2.12: Indirection Buffer Operation

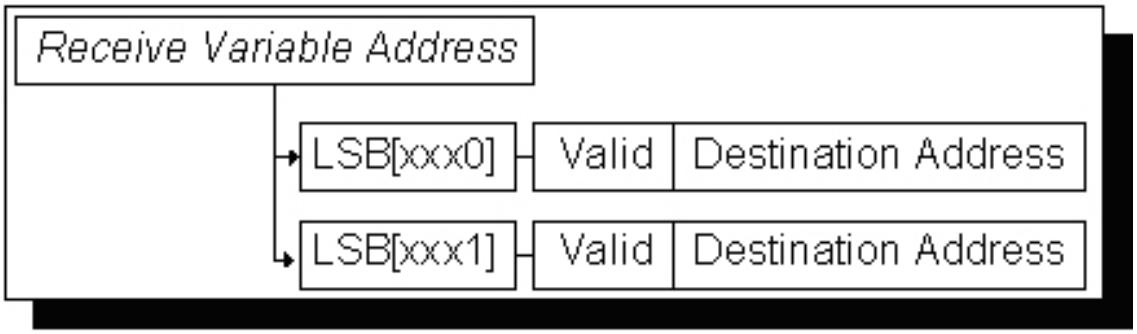

Figure 2.13: Indirection Buffer Structure, 2 Buffers

buffers instructions, because rollbacks would never occur (ie. only proceed when the buffer is known to be correct). However, the synchronization between the SPU processor, the Indirection Buffer, the Indirection Cache, and the scratchpad memory would be a challenge with this option. One possible mechanism would be for all state changes to propogate outwards from the SPU processor. So, for example, if a message is pushed in to one of the receive buffers by another processor, the local spu processor would be notified immediately to update the Indirection Cache to the correct state. The Indirection Buffer is invalidated in this case, because it is simpler and quicker to re-search the cache on the next access and let it become populated naturally. The trivial case is when the receive instructions originate in the spu processor. In this case all that has to be done is to update the Indirection Cache and Indirection Buffer simultaneously with the new information.

The energy saving mechanism is investigated in the results of section 5.3.

Chapter 3

Methodology

3.1 Experimental Setup

The base of the experimental setup is the SimpleScalar simulator [3]. The simulator has been previously modified to accommodate the caching mechanisms from Khunjush's thesis [13]. This meant that the existing instrumentation within the simulator could be used to reproduce the previous results, and further extended to conduct new tests.

It is important to recap how a single core simulator is able to simulate a multicore benchmark. The general approach is to only simulate the root node (node 0) of the MPI application, and then supply it with the correct sequence of messages during the simulation, so it behaves in the correct manner and receives the correct data. The sequence of messages used, are obtained from runs of actual benchmarks, and the full details on how this is accomplished are included later in this section. The choice of the root node as the node of interest was based off of several key facts. For many cases (and in particular for the benchmarks we have chosen), the root node is the manager of all the nodes. This implies the most demanding communication occurs at this node, in the sense that it must send and receive more data to a wider range of nodes. It is also important to note that the current system has the capability to monitor any of the nodes in the system, but the node of interest for this study is the root node, and therefore the additional logging was switched off.

The first part of the process involves choosing appropriate parallel benchmarks to conduct the experiments on. The 64-processor NAS-CG and the PSTSWM benchmarks were chosen by Khunjush for his initial investigations [4] [25]. The experiments

that have been conducted since then used both the 8/9 and 64 processor versions of the NAS-CG, PSTSWM, and the NAS-BT benchmarks. One important feature of all these Benchmarks is their widespread use of the MPI_Send and MPI_Recv functions. Additionally, the size 8/9 benchmarks match the SPU count of the Cell processor, while the larger 64 processor versions reflect the future many core processors.

The second part of this process involves running each of the test benchmarks in an actual parallel environment. Khunjush's work utilized the University of Victoria Minerva supercomputer, which has "128 375-MHz RS/6000 processors and 64 gigabytes of memory" [23] and utilizes a 500MB/second point to point full duplex communication connection on each node. The latest research migrated to use the newest supercomputer at the University of Victoria, the Nestor cluster. This cluster "consists of 288 IBM iDataplex servers with eight 2.67 GHz Xeon x5550 cores and 24 GB of RAM" each networked with high-speed InfiniBand interconnect [22]. The purpose of conducting these runs is to collect the MPI message meta-information and payload data. The meta-information includes the sender, receiver, tag, size, and payload datatype. This second part of the process, of collecting MPI communication traces, uses the MPI Standard Profiling interface, to intercept calls to the desired MPI functions [24]. The custom instrumentation source is compiled as a static library, which can be linked into each of the benchmarks when they are compiled. In this way the desired functions, for example: MPI_Recv or MPI_IRecv and MPI_Wait, are monitored over the entire run of the benchmark. For node 0, the code records the function parameters that have been passed in (ie. sender, receiver, tag, size, and payload datatype), and dumps them to a new line in a trace file (for full details refer to Appendix A). When the message has been received (i.e. MPI_Recv or MPI_Wait unblocks) then the message payload is written out to a separate payload file. The order is synchronized so that reading through the trace file start to end allows one to simultaneously navigate the payload file. This is done by reading the current message from the trace to get the byte length of the message, and then copying or seeking ahead in the payload file by this amount. The two files are assembled in this way because of the nature of the receive operations. That is, for the non-blocking receive calls (MPI_IRecv) the trace file information will be posted immediately, but then the associated data will not be available until the MPI_Wait call completes.

Once the traces and payloads have been collected, the next step is to get the benchmark working on the SimpleScalar simulator. Because the simulator architecture does not match that of the host system, the benchmarks must be cross compiled

using the provided SimpleScalar tools. Only gcc is provided as a cross compiler, and therefore the F2C (Fortran-to-C) tool must be used in conjunction with gcc to compile the test benchmarks (which are in Fortran). Additionally, a faux-MPI version is compiled in with each benchmark. It provides function stubs for all the MPI calls made by each benchmark. Most just return a non-error value (e.g. Benchmark believes it has successfully sent a message, initialized, etc). The MPI_Size is set to return either 8/9 or 64 depending on the benchmark, and the MPI_Rank must be set to return 0 so it becomes the root node. For the MPI_Recv and MPI_IRecv functions a custom assembly instruction is added which is caught by the simulator, and triggers the simulator's architectural extensions. As part of this process, the simulator accesses the trace and payload files to get the messages required by the benchmark, at the appropriate times. In this way, the correct data is also returned to the benchmark, which runs to completion.

The final goal is to use the simulator to obtain a trace pattern of the receive variable accesses during benchmark execution. This can be done by monitoring the cache, tracking each access, and logging the receive variable address and type of access (new receive or data access). More specifically, when messages are received for the first time, the receiving variable address is recorded (and a count incremented if received multiple times), along with the message length in bytes. Then, every time a receiving variable is accessed in the network cache (sized to avoid evictions and ensure logging), the extra logging functions are able to see what address is currently being accessed. This in turn allows the simulator to match the address to one of the received variable addresses, and increment an access count. This entire method is summarized in the Figure 3.1, and the resulting counts are discussed in greater detail in the following section.

3.2 What does the simulator allow us to measure?

The SimpleScalar simulator, was initially modified for Khunjush's thesis. The modifications allow it to reflect the modified caching environment and to receive the simulated sequence of messages, so it behaves like the root node. Because it is not simulating a Cell processor, the receive variable access patterns are of the most importance and have been leveraged to provide three key statisics, which are then used to evaluate the performance of the cacheless architecture approach. To perform this analysis the following must be determined for each receive variable over the course

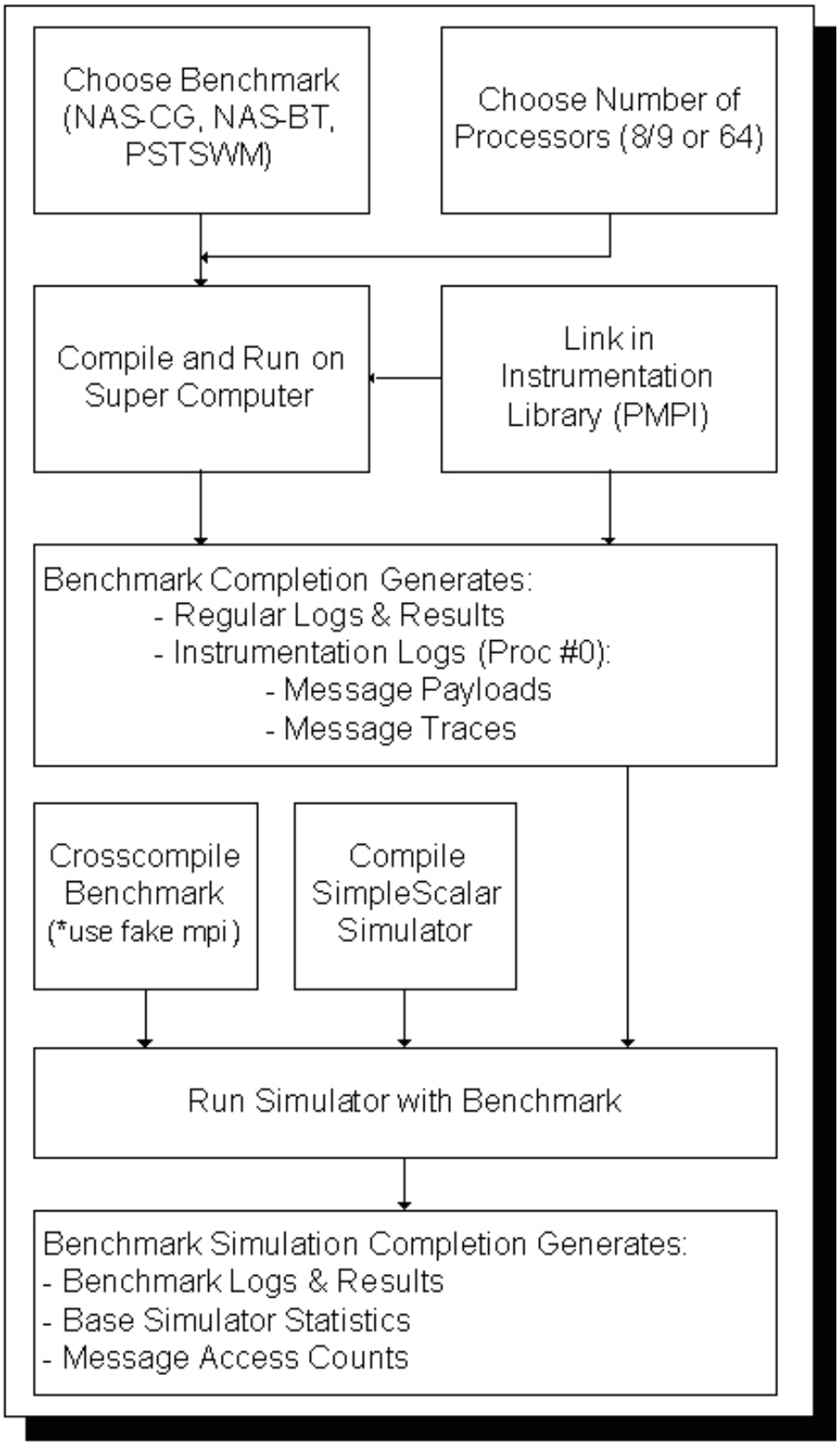

Figure 3.1: Methodology Flowchart

of each benchmark: The classical copying method time cost, the Indirection Cache lookup time cost, and the Indirection Buffer time cost. The classical copying method cost for each variable can be calculated using the number of receives per variable multiplied by the time to transfer a variable of that size. The Indirection Cache lookup time cost can be calculated by multiplying the number of accesses to each variable by the number of cycles to access the cache. Finally, the Indirection Buffer performance can be calculated from the number of mispredictions that occur (ie. the number of times the indirection penalty must still be paid), multiplied by the number of cycles to access the cache. This section examines these calculations in more detail. The following are the values available for each receiving variable after the simulation logging has completed:

- The starting address
- The byte length of the variable
- # of MPI_Recv calls posted for the variable
- # of times the variable was accessed
- # of times hypothetical Indirection Buffer would save time accessing this message (i.e. correct prediction)
- # of times hypothetical Indirection Buffer would not save time (i.e. misprediction)

Totals of each of these values, across all receive variables, are also available at the end of the simulation. They are then used to estimate the performance of the Indirection Buffer caching system, the standard Indirection Cache, and to compare to the classical message-copying time penalty. The estimates are based on the formulas outlined in section 3.3, and all derive from the variable access counts and variable access interleaving patterns. The full calculations and source that has been added to the simulator are contained in Appendix A, but are described briefly in the rest of this section.

As mentioned previously, the extra simulator source code is called when an MPI_Recv is initially received, and for every access to its corresponding receive variable. Tracking the access count values works on the principle that when an MPI_Recv call is made the receive variable address and variable byte-length can be used to determine

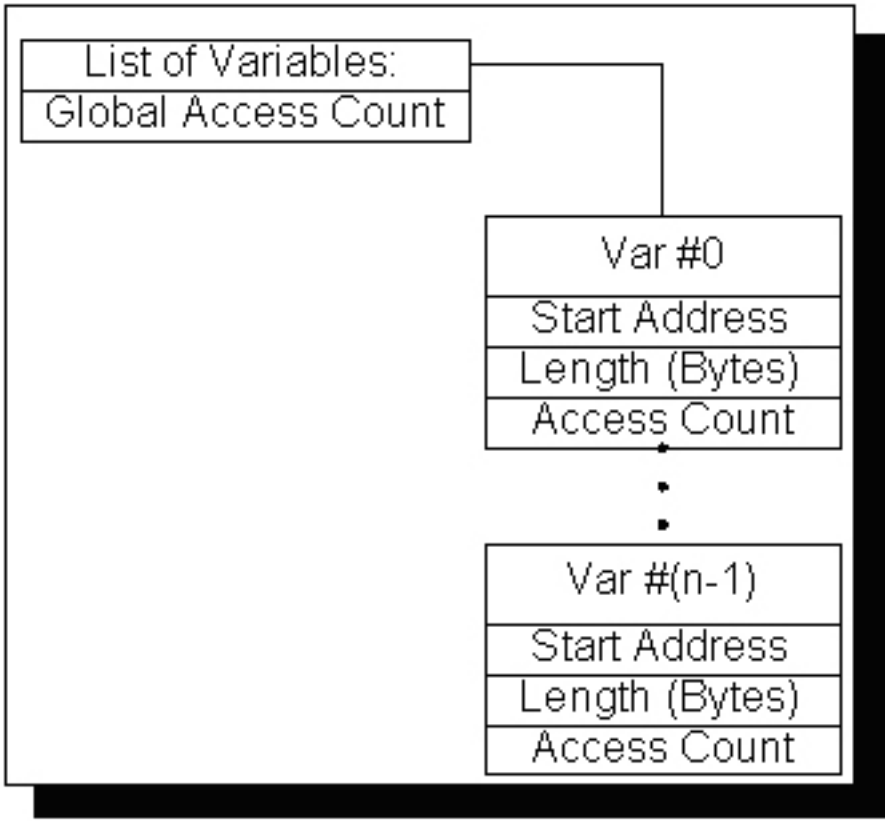

Figure 3.2: Basic Receive Address Access Counting

all accesses to that variable. This is because all accesses will be to addresses that fall within the range: variable address to variable address + length (in bytes).

In addition to giving the MPI_Recv call count and number of accesses, per variable, this mechanism also lets us test the behavior of the Indirection Buffer. Every time an MPI_Recv call comes in it updates an address tracker variable with the current receive variable address. For regular accesses it checks the address tracker, and if it matches, increments a hit counter for the receive variable being accessed. Conversely, if it does not match, it increments a miss counter and then updates the address tracker to the current receive variable.

A more complex mechanism has been implemented which simultaneously tests both a single Indirection Buffer and a double Indirection Buffer implementation. Two separate sets of address trackers, and hit/miss counts are maintained. This allows all of the new experimental data to be collected and calculated from a single simulation run.

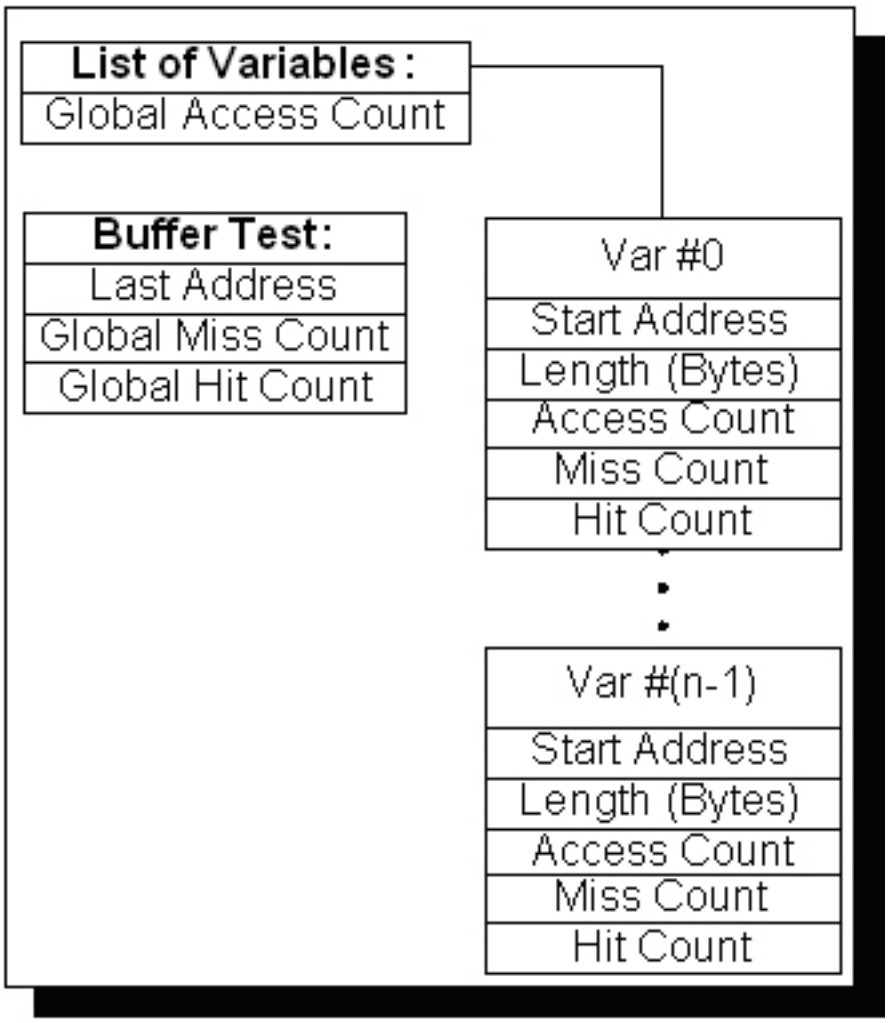

Figure 3.3: Indirection Buffer Counting Diagram

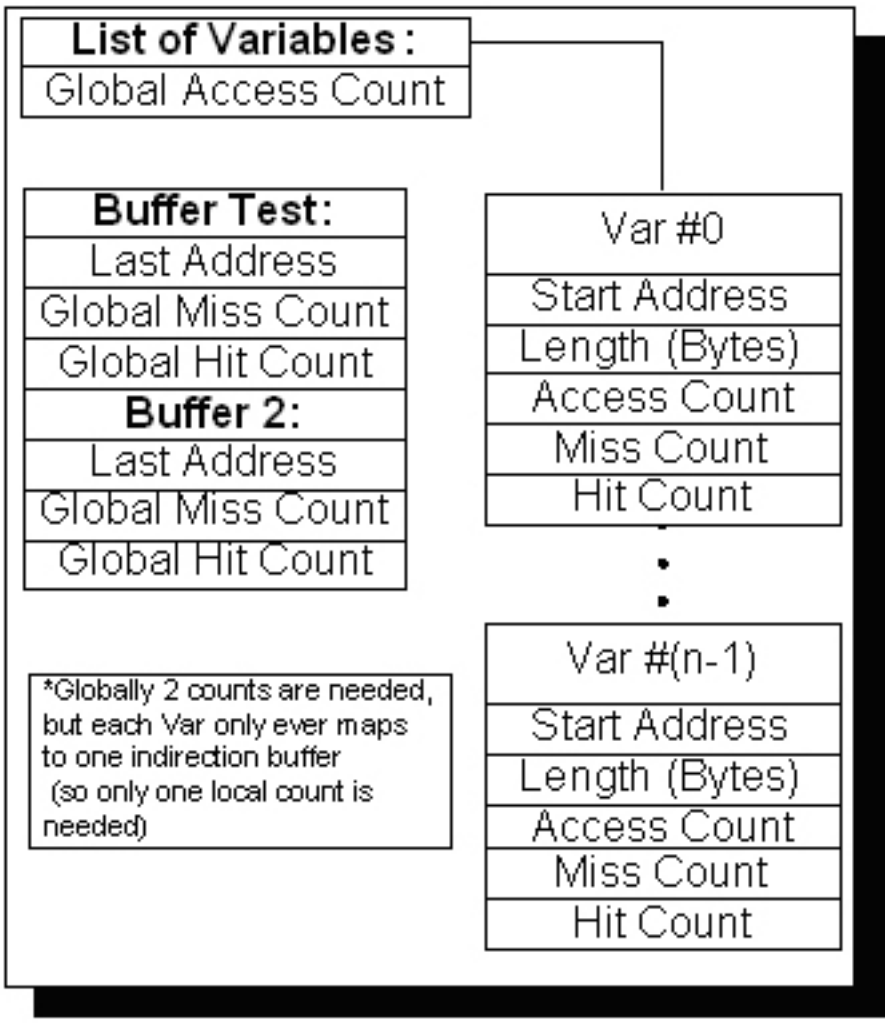

Figure 3.4: Double Indirection Buffer Counting Diagram

An unfortunate dependency of this implementation is its integration with the simulator, and that results are not generated until the simulation has completed. Recall, that the traces are first collected, then fed into the SimpleScalar simulator, and then the simulator tracks the receive variable accesses and upon completion outputs the access totals and statistics. This is problematic because the simulations usually take several hours to run, and in the worst case (BT-9) several days. This makes investigation quite time consuming, especially during code development, and while iterating through many small variations in the final statistics and calculations. Therefore, the solution was to externalize the last portion of the analysis, so it could be done post-simulation. To allow this, the portions of the in-simulation analysis were modified to dump to a log file every time MPI_Recv was initially called, and for every access to a receive variable. Each line in the log file records: If it was an MPI_Recv or a regular access, and the address being accessed. This log can can then be fed into a separate analysis program, which only takes several seconds to complete. In this way, if any changes need to be made to the analysis the program can be quickly re-run on the log file. This works, because the resulting log files use data whose order and values are static from simulation to simulation. (ie. same sequence of receives and memory accesses for a given benchmark). This made additional analysis feasible, whereas previously it was quite time consuming.

3.3 How are results calculated?

Keep in mind that the overall goal is to determine the total time it takes to access each receive variable, and the total time across all of these variables for each benchmark. To achieve this goal, the time for each access to each variable must be calculated. This can be estimated using the size of each variable as part of the basic calculation.

Additionally, the timing calculations for the simulation are done under the assumption that the architectural extensions have been implemented on a Cell Processor. Khunjush used initial measurements on the Cell Processor to create the following Table 3.1[12], which is used for estimating the classical transfer delay. There are various ways to move data around the Cell processor, and the measurements were designed to be as comprehensive as possible. The 'Same SPE' methods are the ones of most interest for the current experiments, because they can be used to calculate how long messages will take to be copied classically between buffers.

To determine the classical message-copying time penalty the following formula is

Method	16B	1KB	4KB	8KB	16KB
PPE-Initiated GET	7.4300	7.4300	7.4300	7.4300	7.4700
SPE-Initiated GET	0.1500	0.1900	0.3900	0.6900	1.2800
PPE-Initiated PUT	7.6000	7.6000	7.6000	7.6000	7.6000
SPE-Initiated PUT	0.1000	0.1500	0.2800	0.4900	0.8900
SPEtoSPE GET	0.0720	0.1100	0.2300	0.3900	0.7100
SPEtoSPE PUT	0.0670	0.1000	0.2300	0.3900	0.7100
Same SPE (DMA)	0.0700	0.0750	0.1000	0.2300	0.7400
Same SPE (COPY)	0.0022	0.0910	0.2800	1.1000	4.2000

Table 3.1: Summary of data transfer for blocking cases (us)

used:
$Time\,Cost = (Message\,Byte\,Length) * (Corresponding\,Message\,Length\,Transfer\,Time)$
This requires matching to Table 3.1 for the best Same SPE transfer time, and interpolating as necessary.
To determine the Indirection Cache access overhead the following formula is used:
$Time\ Cost = (Total\ \#\ of\ accesses) * (Cache\ Access\ Time)$
The justification of this is that every time the Indirection Cache is accessed there is a time cost to search and retrieve the correct destination buffer address. The initial assumption was that this would take 1 CPU Cycle, defined based on the 3.2GHz clock of the Cell Processor.

To determine the new Indirection Buffer's performance the following formulas are used:
$Miss\% = (Total\ \#\ of\ buffer\ mispredictions)/(Total\ \#\ of\ Accesses) * 100$

$Time\ Cost = (Total\ \#\ of\ buffer\ mispredictions) * (Cache\ Access\ Time)$
The justification is that every time the value in the Indirection Buffer switches, it is because the current value is invalid, and has caused an incorrect prediction. Therefore any time this occurs the correct address is not available for the length of time that it takes to access the Indirection Cache.

Chapter 4

Results

4.1 Initial Results

Khunjush's initial results from his paper "Architectural Enhancement for Minimizing Message Delivery Latency on Cache-Less Architectures (e.g., Cell BE)" [12] simulated the 64 processor versions of the benchmarks CG and PSTSWM. These results are included inline below, for comparison, and show the improvement of the indirection penalty versus the classical transfer method (classical transfer being the time it takes to copy the message one additional time).

Variable	Variable Size (B)	Indirection	Classical Transfer
var1	8	9.10E-07	2.75E-06
var2	14000	5.73E-05	3.68E-04
var3	8	0.00E+00	2.75E-06
var4	14000	5.34E-05	1.47E-05
var5	16	1.47E-08	1.06E-07
Sum		1.12E-04	3.88E-04

Table 4.1: Overhead for each Receiving Variable in different Approaches for CG

Variable	Variable Size (B)	Indirection	Classical Transfer
var1	8	7.56E-08	4.16E-07
var2	8	1.52E-07	4.16E-07
var3	128	7.65E-07	5.46E-07
var4	56	1.77E-07	5.54E-07
var5	2048	9.16E-06	1.96E-06
var6	15136	1.83E-05	1.82E-04
var7	1024	4.71E-06	6.37E-07
var8	7568	1.28E-07	6.90E-07
var9	0	3.94E-08	0.00E+00
var10	8192	1.36E-04	3.91E-04
var11	11344	6.71E-05	3.60E-04
Sum		2.36E-04	9.38E-04

Table 4.2: Overhead for each Receiving Variable in different Approaches for PSTSWM

4.2 New Results - One Cycle Assumption

The new research has expanded the tested benchmarks to include the following (name-# processors): BT-9, BT-64, CG-8, CG-64, PSTSWM-9, & PSTSWM-64. This provides two sets of benchmarks to compare, namely those of 8/9 processor size and those of 64 processor size. Due to refinements in the calculation techniques, the values for the classical transfer times are slightly different than the ones reported in Section 4.1. This is because the refinements use the fastest of the DMA or COPY values in Table 3.1, and approximate off the closest matching measured data size. This is to give a lower bound for the classical transfer time, when comparing to the calculated indirection penalty. These initial results also continue with the assumption that the Cache Access Time is one cycle.

To recap, the time in seconds is the unit of comparison for each of the methods in the tables. The classical transfer time is the length of time it takes to copy each message, and is broken down by receive variable but summed over the course of the benchmark. The indirection time is based on the number of acceses each receive variable has, multiplied by the penalty for each of these accesses. The New 1 and 2 buffer solutions are based on the number of receive variable accesses which would miss, multiplied by the penalty for each of these accesses.

4.2.1 Results Agenda

The following sections contain: tables which show the timing costs (in seconds) for each receive variable, for each benchmark tested, the miss rates of the new buffers, and also summarize the results graphically. These results are then discussed in **Section 4.2.6**. The timing costs represent the aggregate time spent on the given operation (Classical transfer operation, indirection penalty, or reduced indirection penalty because of buffering), over the course of the benchmark.

Section 4.2.2 8/9 Processor Benchmarks: Tables for each, graph of miss % of Indirection Buffer, graph comparing Indirection Cache and Buffer approaches.

Section 4.2.3 64 Processor Benchmarks: Tables for each, graph of miss % of Indirection Buffer, graph comparing Indirection Cache and Buffer approaches.

Section 4.2.4 8/9 Processor Improvement from Classical (x): Tables for each benchmark, Summary table, Summary graph.

Section 4.2.5 64 Processor Improvement from Classical (x): Tables for each benchmark, Summary table, Summary graph.

4.2.2 8/9 Processor Benchmarks

Address	Size	Classical Transfer	Indirection	New (1 buffer)	New (2 buffer)
var 1	74000	1.350E-03	6.830E-04	1.008E-06	1.008E-06
var 2	70560	6.438E-04	1.198E-03	1.385E-06	1.071E-06
var 3	74000	1.350E-03	1.461E-04	6.313E-07	6.313E-07
var 4	74000	6.751E-04	1.460E-04	6.313E-07	6.313E-07
var 5	116160	1.265E-02	1.825E-03	7.538E-07	7.538E-07
Sum		1.667E-02	3.998E-03	4.409E-06	4.095E-06

Table 4.3: Time Costs (in seconds) for each Receive Variable in different Approaches for BT-9

Address	Size	Classical Transfer	Indirection	New (1 buffer)	New (2 buffer)
var 1	8	1.830E-06	2.600E-07	2.600E-07	2.600E-07
var 2	28000	1.518E-03	1.139E-04	3.797E-07	3.797E-07
var 3	8	1.830E-06	3.900E-07	2.650E-07	2.600E-07
var 4	28000	6.070E-05	1.091E-04	1.369E-07	1.322E-07
var 5	16	7.040E-08	3.125E-10	3.125E-10	3.125E-10
Sum		1.582E-03	2.236E-04	1.042E-06	1.032E-06

Table 4.4: Time Costs (in seconds) for each Receive Variable in different Approaches for CG-8

Address	Size	Classical Transfer	Indirection	New (1 buffer)	New (2 buffer)
var 1	8	5.280E-08	3.125E-10	3.125E-10	3.125E-10
var 2	8	1.760E-08	7.875E-08	7.781E-08	7.656E-08
var 3	56	2.464E-07	9.194E-07	4.563E-07	3.809E-07
var 4	15136	3.336E-04	1.536E-04	4.709E-06	4.709E-06
var 5	7568	1.848E-07	8.293E-05	4.634E-06	4.634E-06
var 6	60544	1.329E-03	2.840E-04	1.519E-07	1.519E-07
var 7	0	0.000E+00	7.594E-08	7.594E-08	7.594E-08
var 8	0	0.000E+00	7.594E-08	7.594E-08	7.594E-08
Sum		1.663E-03	5.217E-04	1.018E-05	1.011E-05

Table 4.5: Time Costs (in seconds) for each Receive Variable in different Approaches for PSTSWM-9

Name - #buffers	Mis-predictions	Total Accesses	Percent Misses	Indirection Cost	New Cost
BT - 1	1.411E+04	1.279E+07	0.110	3.998E-03	4.409E-06
BT - 2	1.311E+04	1.279E+07	0.102	3.998E-03	4.095E-06
CG - 1	3.334E+03	7.154E+05	0.466	2.236E-04	1.042E-06
CG - 2	3.303E+03	7.154E+05	0.462	2.236E-04	1.032E-06
PSTSWM - 1	3.258E+04	1.670E+06	1.952	5.217E-04	1.018E-05
PSTSWM -2	3.234E+04	1.670E+06	1.937	5.217E-04	1.011E-05

Table 4.6: Benchmark Side-by-side Buffer Performance Comparison for 8/9 Processors

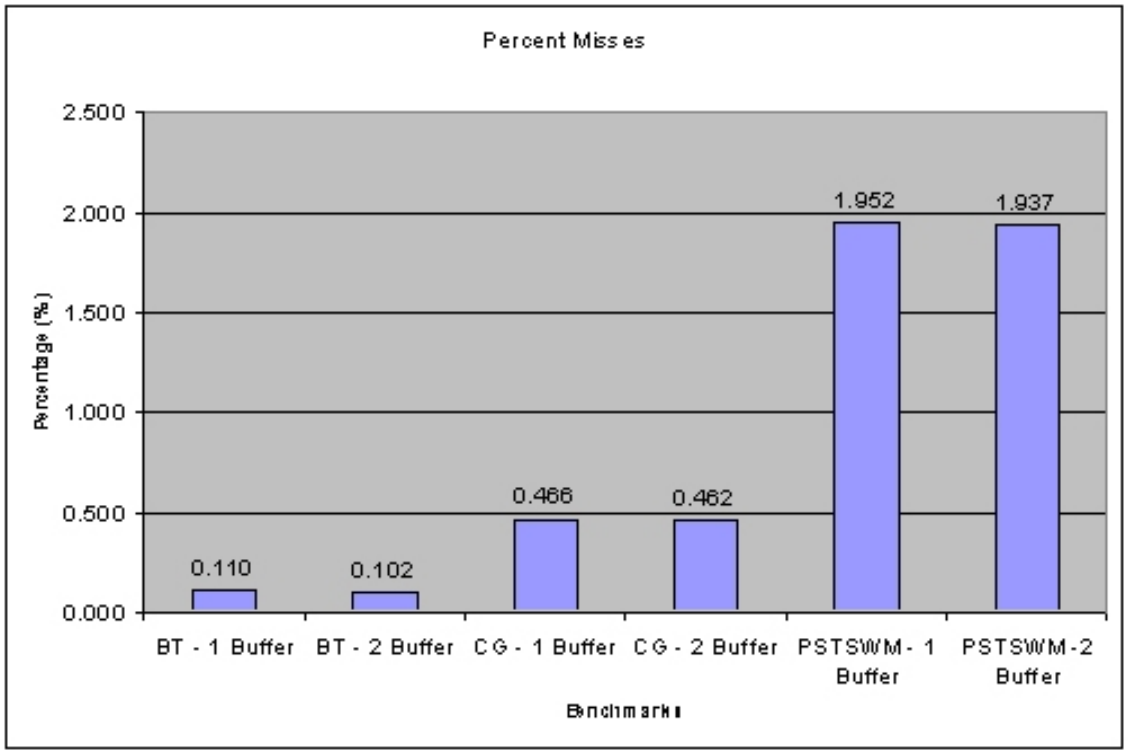

Figure 4.1: Average Miss % of 8/9 Processor Benchmarks, when Indirection Buffer is used (see section 3.3)

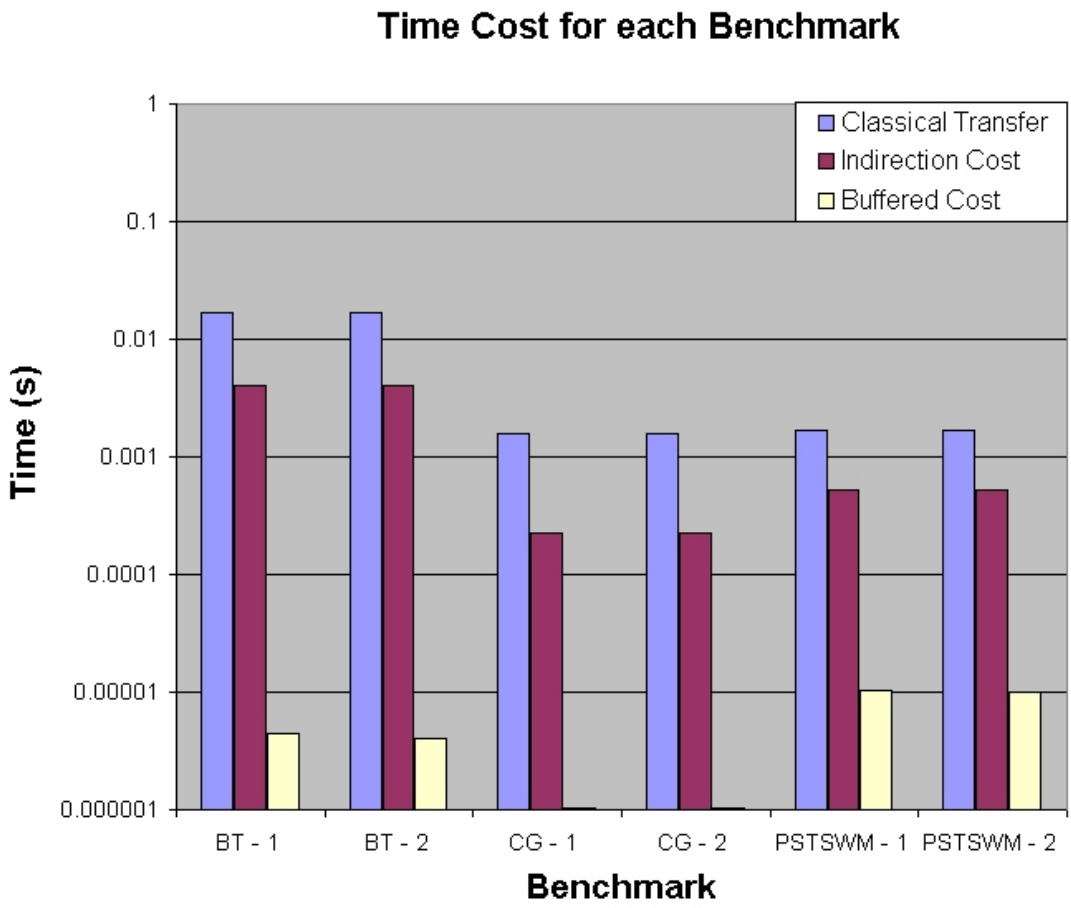

Figure 4.2: Time Cost (in seconds, & Logarithmic) of 8/9 Processor Benchmarks

4.2.3 64 Processor Benchmarks

Address	Size	Classical Transfer	Indirection	New (1 buffer)	New (2 buffer)
var 1	35840	6.540E-04	7.070E-05	2.209E-06	2.209E-06
var 2	35840	3.270E-04	8.922E-04	3.968E-06	2.272E-06
var 3	35840	6.540E-04	7.070E-05	2.209E-06	2.209E-06
var 4	35840	3.270E-04	7.070E-05	2.209E-06	2.209E-06
var 5	19440	7.412E-03	8.600E-04	2.638E-06	2.638E-06
Sum		9.374E-03	1.964E-03	1.323E-05	1.154E-05

Table 4.7: Time Costs (in seconds) for each Receive Variable in different Approaches for BT-64

Address	Size	Classical Transfer	Indirection	New (1 buffer)	New (2 buffer)
var 1	8	2.746E-06	9.100E-07	5.200E-07	3.950E-07
var 2	14000	1.012E-03	5.730E-05	5.047E-07	5.000E-07
var 3	8	2.746E-06	0.000E+00	0.000E+00	0.000E+00
var 4	14000	4.047E-05	5.342E-05	1.372E-07	1.372E-07
var 5	16	1.056E-07	0.000E+00[1]	0.000E+00[1]	0.000E+00[1]
Sum		1.058E-03	1.116E-04	1.162E-06	1.032E-06

Table 4.8: Time Costs (in seconds) for each Receive Variable in different Approaches for CG-64

[1] 0 time cost is associated with variables that are not accessed, only received. For example, messages used for synchronization can be facilitated by a receive call, and therefore the data is unimportant and never accessed because all that matters is the message arrival.

Address	Size	Classical Transfer	Indirection	New (1 buffer)	New (2 buffer)
var 1	8	4.158E-07	3.125E-10	3.125E-10	3.125E-10
var 2	8	1.386E-07	7.688E-08	7.625E-08	7.625E-08
var 3	128	5.625E-08	7.653E-07	1.534E-07	1.534E-07
var 4	56	1.940E-06	1.594E-07	1.544E-07	1.544E-07
var 5	2048	1.050E-06	9.145E-06	9.894E-07	9.894E-07
var 6	15136	1.682E-04	1.826E-05	7.688E-08	7.688E-08
var 7	1024	5.250E-07	4.707E-06	9.897E-07	9.897E-07
var 8	7568	2.771E-07	1.284E-07	9.375E-10	9.375E-10
var 9	0	0.000E+00[2]	7.594E-08	7.594E-08	7.594E-08
var 10	8192	3.912E-04	1.357E-04	5.316E-07	5.316E-07
var 11	11344	2.490E-04	6.702E-05	2.275E-07	1.522E-07
Sum		8.131E-04	2.361E-04	3.352E-06	3.277E-06

Table 4.9: Time Costs (in seconds) for each Receive Variable in different Approaches for PSTSWM-64

[2]The 0 byte size is an exception case which manifests itself for receive variables used in non-standard ways. The presence in the results is worth noting, but has negligible impact on the overall result.

Name - #buffers	Mis-predictions	Total Accesses	Percent Misses	Indirection Cost	New Cost
BT - 1	4.235E+04	6.286E+06	0.674	1.964E-03	1.323E-05
BT - 2	3.692E+04	6.286E+06	0.587	1.964E-03	1.154E-05
CG - 1	3.718E+03	3.572E+05	1.041	1.116E-04	1.162E-06
CG - 2	3.303E+03	3.572E+05	0.925	1.116E-04	1.032E-06
PSTSWM - 1	1.073E+04	7.555E+05	1.420	2.361E-04	3.352E-06
PSTSWM -2	1.049E+04	7.555E+05	1.388	2.361E-04	3.277E-06

Table 4.10: Benchmark Side-by-side Buffer Performance Comparison for 64 Processors

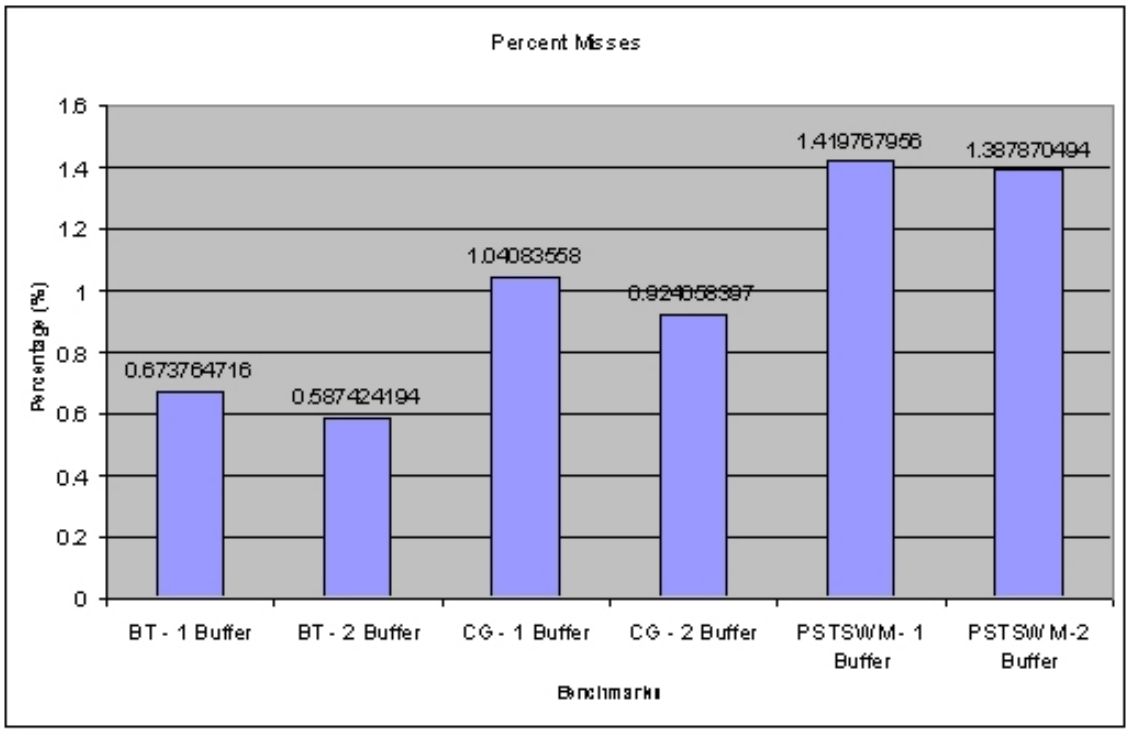

Figure 4.3: Average Miss % of 64 Processor Benchmarks, when Indirection Buffer is used (see section 3.3)

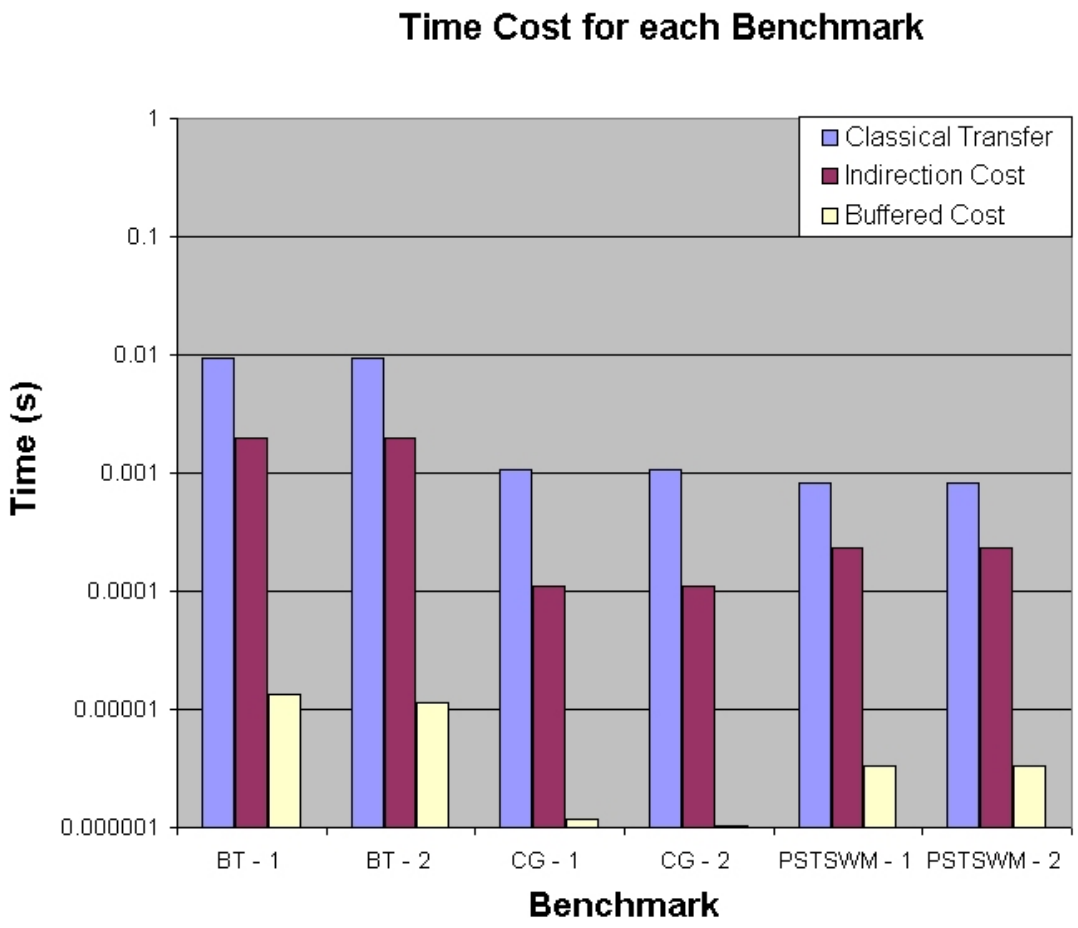

Figure 4.4: Time Cost (in seconds, & Logarithmic) of 64 Processor Benchmarks

4.2.4 8/9 Processor Benchmarks Improvement from Classical

Address	Size	Classical Transfer	Indirection	New (1 buffer)	New (2 buffer)
var 1	74000	1	1.98	1339.40	1339.40
var 2	70560	1	0.54	464.81	601.12
var 3	74000	1	9.24	2139.06	2139.06
var 4	74000	1	4.62	1069.53	1069.53
var 5	116160	1	6.93	16788.75	16788.75
Sum		1	4.17	3781.48	4071.48

Table 4.11: Improvement from Classical for BT-9

Address	Size	Classical Transfer	Indirection	New (1 buffer)	New (2 buffer)
var 1	8	1	7.04	7.04	7.04
var 2	28000	1	13.33	3996.91	3996.91
var 3	8	1	4.69	6.91	7.04
var 4	28000	1	0.56	443.49	459.22
var 5	16	1	225.28	225.28	225.28
Sum		1	7.08	1518.43	1532.68

Table 4.12: Improvement from Classical for CG-8

Address	Size	Classical Transfer	Indirection	New (1 buffer)	New (2 buffer)
var 1	8	1	168.96	168.96	168.96
var 2	8	1	0.22	0.23	0.23
var 3	56	1	0.27	0.54	0.65
var 4	15136	1	2.17	70.84	70.84
var 5	7568	1	0.00	0.04	0.04
var 6	60544	1	4.68	8750.50	8750.50
var 7	0	0	0.00	0.00	0.00
var 8	0	0	0.00	0.00	0.00
Sum		1	3.19	163.34	164.59

Table 4.13: Improvement from Classical for PSTSWM-9

Benchmark	Improvement from Classical (x)
BT - 1	3781.48
BT - 2	4071.48
CG - 1	1518.43
CG - 2	1532.68
PSTSWM - 1	163.34
PSTSWM - 2	164.59

Table 4.14: Summary of 8/9 Processor Benchmarks Improvement from Classical

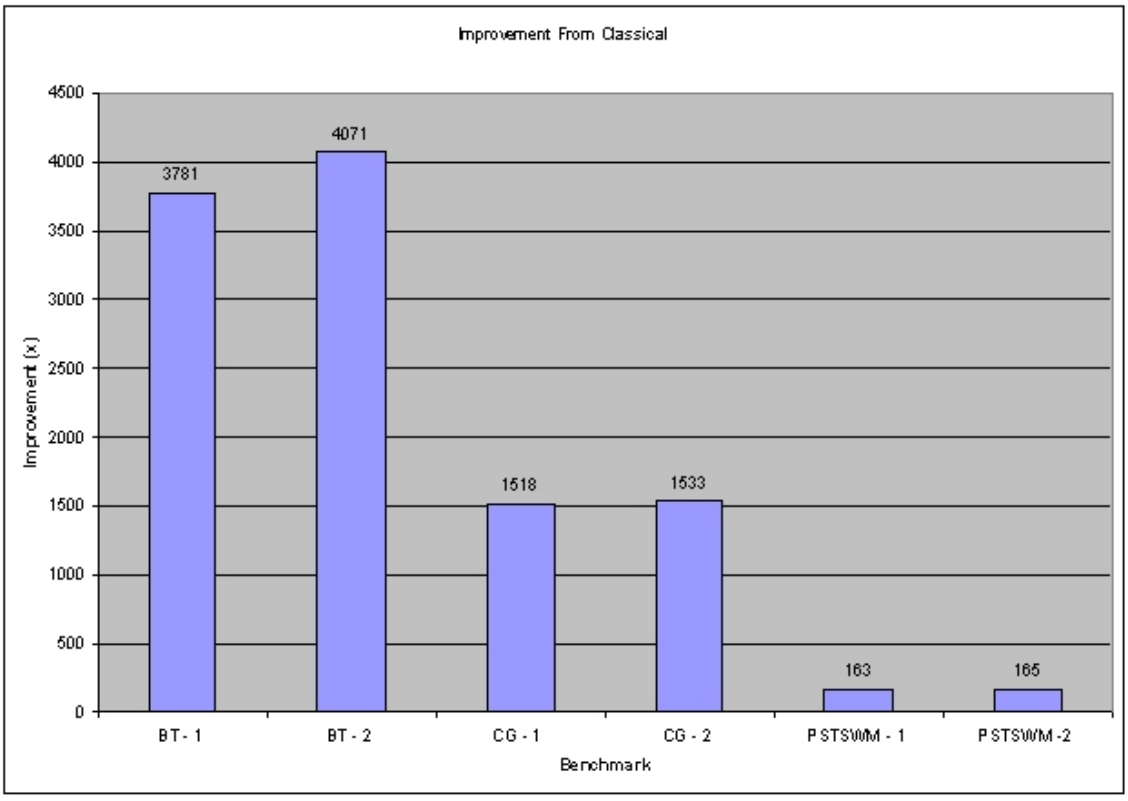

Figure 4.5: 8/9 Processor Benchmarks Improvement from Classical

4.2.5 64 Processor Benchmarks Improvement from Classical

Address	Size	Classical Transfer	Indirection	New (1 buffer)	New (2 buffer)
var 1	35840	1	9.25	296.00	296.00
var 2	35840	1	0.37	82.40	143.91
var 3	35840	1	9.25	296.00	296.00
var 4	35840	1	4.63	148.00	148.00
var 5	19440	1	8.62	2809.69	2809.69
Sum		1	4.77	708.32	812.44

Table 4.15: Improvement from Classical for BT-64

Address	Size	Classical Transfer	Indirection	New (1 buffer)	New (2 buffer)
var 1	8	1	3.02	5.28	6.95
var 2	14000	1	17.66	2004.64	2023.44
var 3	8	1	0.00	0.00	0.00
var 4	14000	1	0.76	294.99	294.99
var 5	16	1	0.00	0.00	0.00
Sum		1	9.48	910.41	1024.80

Table 4.16: Improvement from Classical for CG-64

Address	Size	Classical Transfer	Indirection	New (1 buffer)	New (2 buffer)
var 1	8	1	1330.56	1330.56	1330.56
var 2	8	1	1.80	1.82	1.82
var 3	128	1	0.07	0.37	0.37
var 4	56	1	12.18	12.57	12.57
var 5	2048	1	0.11	1.06	1.06
var 6	15136	1	9.21	2187.63	2187.63
var 7	1024	1	0.11	0.53	0.53
var 8	7568	1	2.16	295.63	295.63
var 9	0	0	0.00	0.00	0.00
var 10	8192	1	2.88	736.00	736.00
var 11	11344	1	3.72	1094.54	1636.20
Sum		1	3.44	242.56	248.13

Table 4.17: Improvement from Classical for PSTSWM-64

Benchmark	Improvement from Classical (x)
BT - 1	708.32
BT - 2	812.44
CG - 1	910.41
CG - 2	1024.80
PSTSWM - 1	242.56
PSTSWM -2	248.13

Table 4.18: Summary of 64 Processor Benchmarks Improvement from Classical

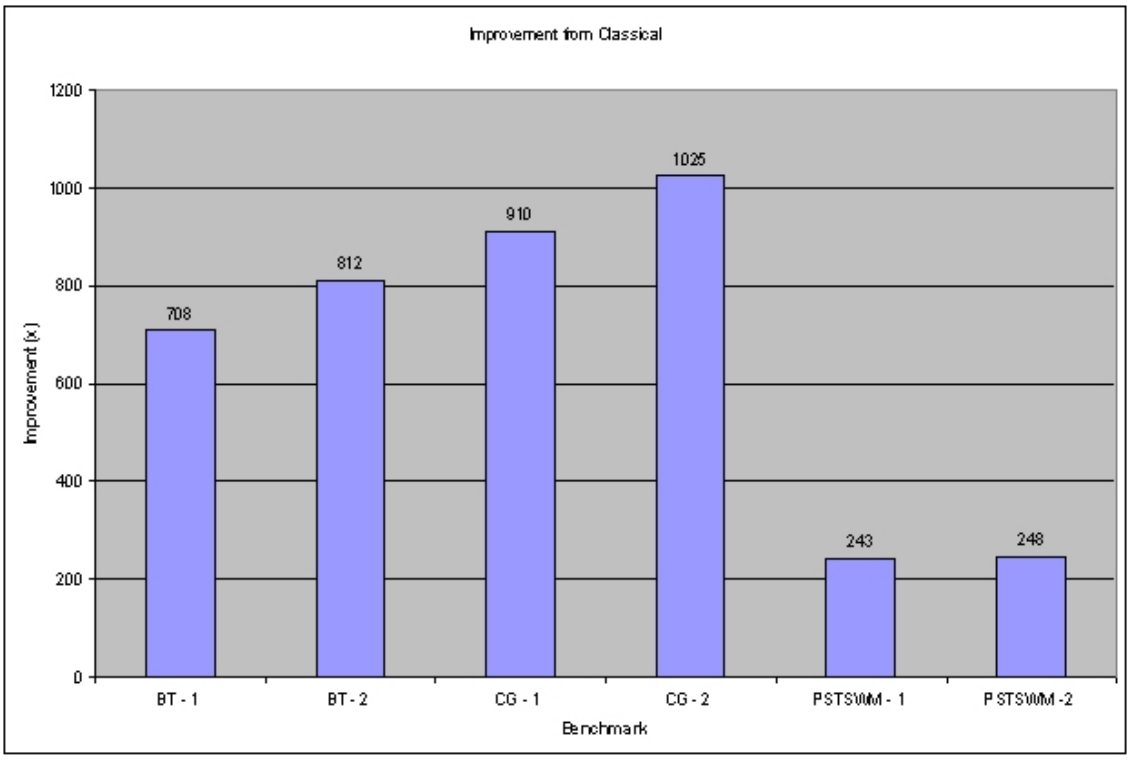

Figure 4.6: 64 Processor Benchmarks Improvement from Classical

4.2.6 Analysis

Looking at the timing tables, it can be seen that there are still some variables that have a classical copying less than the indirection cost. Examples show up only for several pstswm variables (ie. Var2, Var3, and Var5 in Table 4.5, and Var3, Var5, and Var7 in Table 4.9), as this benchmark has more small/medium size variables and a larger number of accesses per variable receive. PSTSWM also has the highest miss %, for the Indirection Buffer, which means that for certain variables there is less of a reduction in access time. That said, the overall Indirection Buffer Miss percentages are all quite low, less than 2%, across all benchmarks and all sizes. This means that the Indirection Buffer is able to minimize the Indirection Penalty and therefore reduce the access times.

This is excellent performance and the overall time cost drops by 2-3 orders of magnitude for the 8/9 processor benchmarks. The Indirection Buffered 8/9 processor benchmarks (BT,CG,PSTSWM) showed large improvements over the Indirection Cache (976x, 215x, and 52x), while the 64 processor benchmark results were generally lower (170x,108x,72x). For the 64 processor benchmarks the improvement is still excellent, at 2 orders of magnitude time cost reduction. The reason for this difference can be attributed to the larger receive variables (greater than 4k, and often into the 100ks) used by the 8/9 processor benchmarks, which means that iterating over all the items in the variable will yield roughly up to 8-10x (approximately proportional to the largest variable size increase) the number of hits in a row. In the same sense, larger receive variables tend to benefit from this the most, as there are many more accesses all hitting the same root receive variable address. This pushes the improvement much lower than for shorter messages (less than or equal to 4k) which have a reduced number of accesses. The general trends of this can be seen when comparing the 8/9 processor benchmarks to the 64 processor benchmarks, and also when comparing the BT benchmark (which has larger messages) to the other benchmarks. The BT benchmarks large variable sizes (even for the 64 Processor run) help to explain the reason its improvements are essentially the same for both 8/9 and 64 sizes. If the CG and PSTSWM benchmarks are examined more closely (BT is omitted from this comparison because all of its variables are considered large) it can be seen that the Indirection Buffer misses much more for the smaller variables (See Tables 4.19 and 4.20). This performance can be attributed to fewer accesses in a row to a single small variable, and more switching between multiple small variables. However, because of

the vast number of accesses to the larger variables, the overall effect is that the average miss percentage is still very low. The result of this effect can be seen, for the single Indirection Buffer solution, in Figure 4.7 which shows the improvement from classical for small and large variables, and overall. Figure 4.8 shows the same results for the Two Indirection Buffer option.

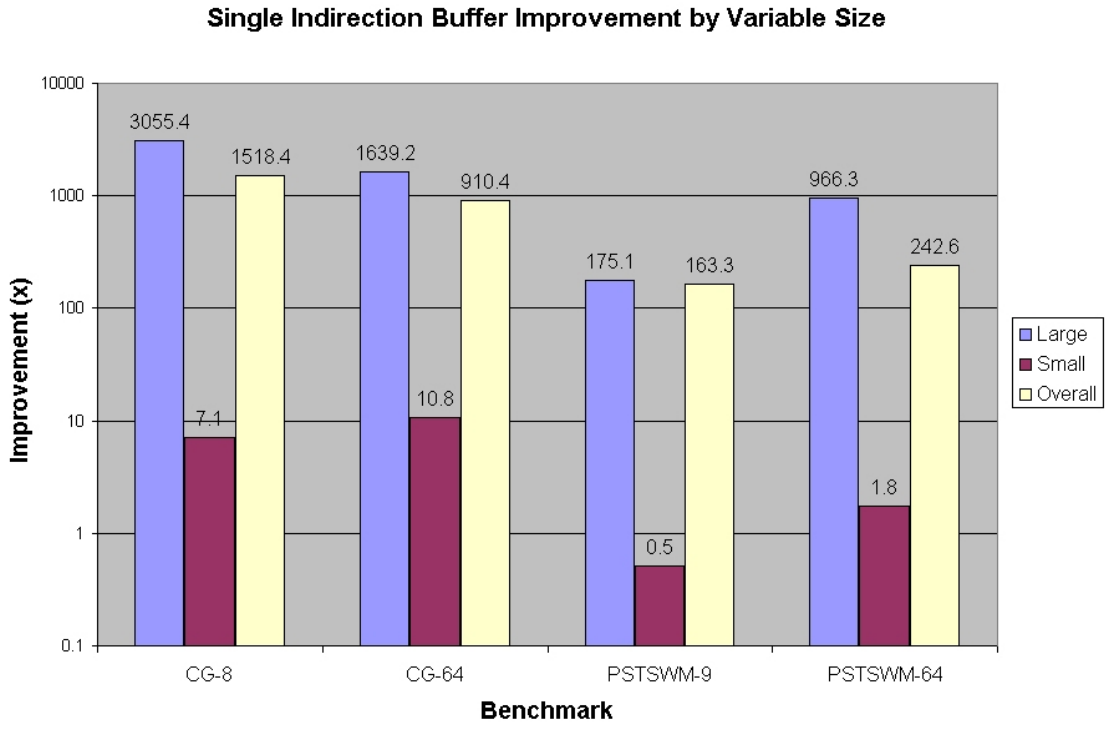

Figure 4.7: Single Indirection Buffer Improvement from Classical, by Variable Size (Logarithmic)

An important observation is that for large variables, adding a second Indirection Buffer provides negligible improvement over the single buffer solution. This indicates that the remaining buffer mispredictions occur because of a new MPI_Recv call being issued, or that the interleaving of receive variable accesses is a more complex pattern (which would require more than two Indirection Buffers to mitigate). It also is a matter of diminishing returns, in other words we already have very high hit percentages and it will likely be difficult to get any closer to 100%.

Where multiple Indirection Buffers provide the most potential benefit, is for the small variable accesses. It can be seen in Tables 4.19 & 4.20 that the small variables' miss percentage drops much more than for the large variables. Unfortunately, the

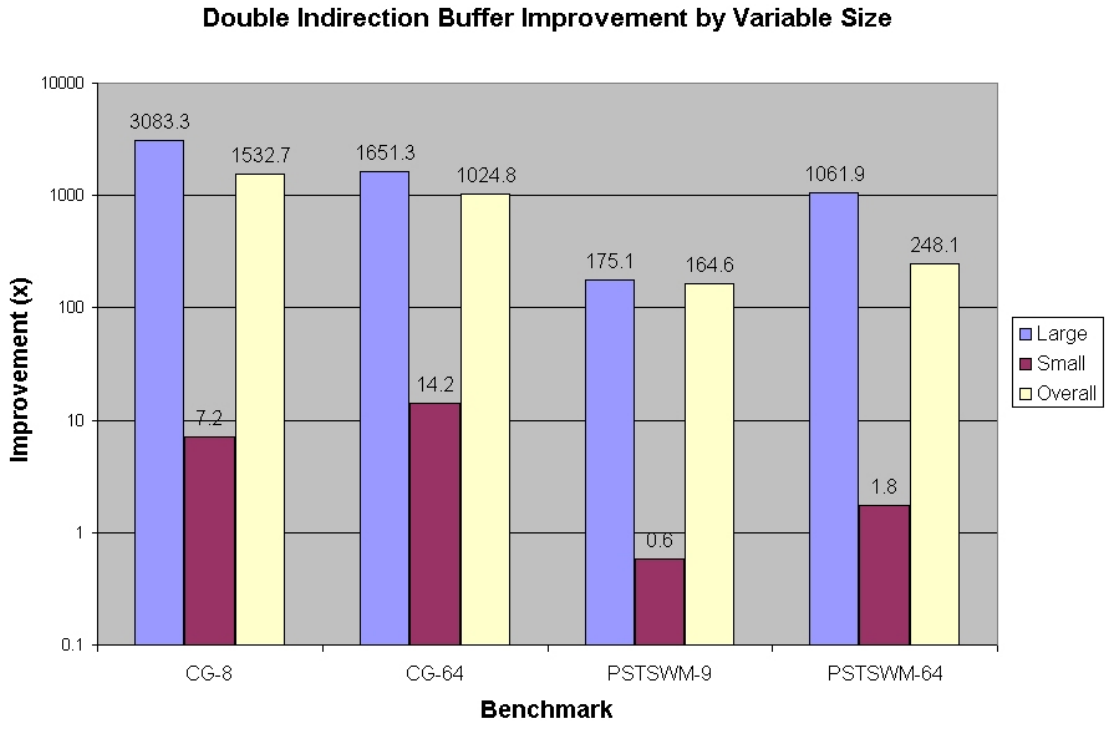

Figure 4.8: Double Indirection Buffer Improvement from Classical, by Variable Size (Logarithmic)

Benchmark	Large	Small	Overall
CG-8	0.23	80.78	0.47
CG-64	0.58	57.14	1.04
PSTSWM-9	1.82	59.67	1.95
PSTSWM-64	0.38	16.76	1.42

Table 4.19: Single Indirection Buffer Miss Percentages

Benchmark	Large	Small	Overall
CG-8	0.23	80.01	0.46
CG-64	0.58	43.41	0.92
PSTSWM-9	1.82	53.01	1.94
PSTSWM-64	0.34	16.76	1.39

Table 4.20: Double Indirection Buffer Miss Percentages

miss percentages are still quite high for small variables (Figure 4.9). Specifically, it is worth looking at PSTSWM-9, in Figure 4.8, where it can be seen that the

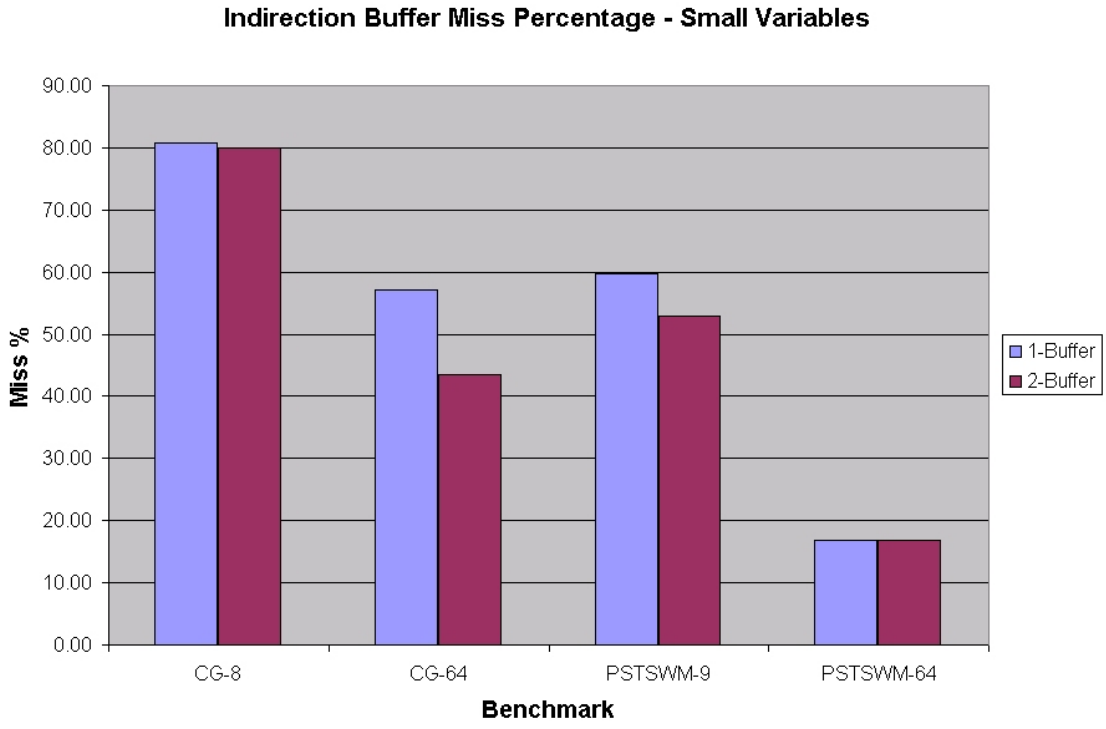

Figure 4.9: Indirection Buffer Miss Percentage - Small Variables

two buffer case provides a slight improvement over the single buffer case, but not enough to push it past the 1x break-even mark. This is due to the behaviour of small variables. In general, accesses to small variables are more interleaved and also much more random, than for the large variables (which tend to be accessed sequentially and with less interleaving). This makes the small variables sensitive to the method of partitioning (the variables) between the two Indirection Buffers. Preliminary testing has shown that for certain partitioning configurations the PSTSWM-9 small variable improvement can reach 1.15x improvement over classical. However, more thorough testing of the partitioning method and number of Indirection Buffers will need to be conducted, to fully establish the benefit for small variable accesses.

The final point to drive home is from Sections 4.2.4 and 4.2.5, which represent and summarize the results, as improvement over the Classical Transfer Method. It can be seen that for just the Indirection Cache the overall improvement ranges from 3.2x to 9.5x. This is reasonable, but for the Indirection Buffer the best case performance (BT-9) is a 4000x improvement, while the worst case (PSTSWM-9) shows a 160x improvement. This gives plenty of incentive to use the Indirection Buffer(s). In

addition, these estimates which use the base single cycle access time assumption for the Indirection Cache, provide plenty of cushion for the Cache Design Investigation in the following Chapter.

Chapter 5

Cache Design Investigation

5.1 Methodology

With the results showing how well the Indirection Cache and Indirection Buffer perform, it now becomes important to look more closely and see if the Indirection Buffer is just an extra improvement or necessary for the architecture to be viable. As part of this, the design parameters of the Indirection Cache need to be examined more closely. Section 5.1.1 does this by looking more precisely at the type and size of Cache used for the Indirection Cache. These design choices effect the timing of the Indirection Cache, which in turn impact its improvement over the Classical Method. The results and analysis of the Cache design are presented in Section 5.2. The goal of this Chapter is to demonstrate that the Indirection Buffer is not only a vast improvement over the standalone Indirection Cache, it is also needed as part of any practical implementation.

5.1.1 Cache Parameters

The two Time Cost calculations, introduced in Section 3.3, depend on Cache Access Time for their result. As a starting point for the investigations in this and previous work, the value was assumed to be 1 clock cycle. This was based on the assumption that the Indirection Cache was fully associative, and 1 clock cycle was presumed to be a reasonable performance estimate. To get more precise results, the value of the Cache Access Time can be set to a time calculated from CACTI [7]. CACTI is a cache performance estimation tool, that allows the user to configure the cache parameters and then obtain many key operating values. In this case, the access time

and energy per access are of the most interest. The CACTI cache tool was configured for several sizes, so that the performance under certain restrictions could be observed. The parameters held constant for each test are listed in Table 5.1. Therefore, beyond these values, all that must be determined is the cache size ranges to test latency over.

Parameter	Justification
90nm	Technology The Cell processor is made with this level.
2-Way Set Associative	standard
64 bit tag	Receive Variable (Addresses are 64 bit)
128 bit input/output bus width	Width of Cell's bus to access local memory
256 bit line size	2 valid bits
	1 release bit
	64 bits for Destination Address
	64 for size mask/message size
	125 for message envelope (tag, src id, comm, etc.)

Table 5.1: CACTI Parameters Table

Continuing under the assumption that the Indirection Cache is fully associative it is possible to simulate the access patterns for each of the benchmarks. From this it is possible to determine the miss rate of a fully associative Indirection Cache, for different sizes. The results obtained are shown in Table 5.2. They show that even for a 2-entry cache the miss rates across all benchmarks were below 0.42%. Again, this is because of many accesses to large variables, which are kept in the cache because of the least recently used replacement policy. This means that the misses are for the few times that smaller variables must be cycled out of the cache. The results also showed that for an 8-entry cache the BT and CG benchmarks recorded no misses, and that the limiting factor is PSTSWM-64 (with its 36 different receive variables, causing additional collisions).

Benchmarks	2-Entry	4-Entry	8-Entry	16-Entry
BT-9	0.00068	0.00000	0.00000	0.00000
BT-64	0.00421	0.00000	0.00000	0.00000
CG-8	0.00059	0.00002	0.00000	0.00000
CG-64	0.00226	0.00008	0.00000	0.00000
PSTSWM-9	0.00248	0.00087	0.00015	0.00000
PSTSWM-64	0.00323	0.00322	0.00322	0.00064

Table 5.2: Fully Associative Cache Miss Ratios

It is also possible to determine the time penalty each of these configurations would incur, because each time there is a miss the missing entry must be moved back into the cache, at a cost of 7 cycles per 128 bits (the expense of accessing Local Storage). This means that for the worst case memory performance (non-banked memory), it would take 14 cycles to reload an entire cache line. When comparing this penalized single cycle Indirection Cache to the original single cycle Indirection Cache in Table 5.3, which shows how many times slower the penalized implementation would be, we can see the 16-Entry is the safest solution. However it is also possible to see that even the 2-Entry has at most a 1.058x slowdown (ie. 6%) from the original Indirection Cache.

Benchmarks	2-Entry	4-Entry	8-Entry	16-Entry
BT-9	1.00949	1.00000	1.00000	1.00000
BT-64	1.05894	1.00000	1.00000	1.00000
CG-8	1.00824	1.00029	1.00000	1.00000
CG-64	1.03159	1.00118	1.00000	1.00000
PSTSWM-9	1.03469	1.01224	1.00204	1.00000
PSTSWM-64	1.04519	1.04512	1.04503	1.00902

Table 5.3: Penalized Indirection Cache Slowdown

The size of the cache at minimum (based on the current benchmarks) would be 2 entries. This gives a cache size of 64 Bytes, and would provide reasonable performance in terms of miss overheads. A good upper bound for the cache size is 64 entries (almost double the 35 maximum unique receive variables for PSTSWM-64). This would give a total cache size of 2048 Bytes. Therefore, to summarize, that gives a range of sizes to test the latency in CACTI over. The results from these tests will show the expected number of cycles (Cache Access Time). The miss overheads will also be presented and factored into the results to show the tradeoff when choosing lower latency, smaller-size caches.

5.2 Cache Cycle-Timing Sweep

The assumption that the Indirection Cache took only one cycle to access, required further investigation using the CACTI tool. The results of the CACTI calculations are included below, and then followed by revised timing calculations for the Indirection Cache and Indirection Buffer approaches.

The CACTI cache simulator has limitations for caches smaller than approximately 32 lines (for 32 Bytes per line that is 1kB total size). The CACTI results presented in figure 5.1 show the resulting timing values for the cache over the range 64B to 4kB, and the timings of the range of interest (2 lines to 64 lines) are presented in Table 5.4. Extrapolation is performed to get timing estimates for the sizes less than 1kB, but only the values from the range 1kB to 1.75kB are used for this linear calculation. This is because there are steps in the performance at certain thresholds that split the cache sizes into linear sets. The most important feature of this graph is that it shows the fastest possible cache is clearly going to have 2-Cycle access time at the very best, and 1-Cycle access is not possible. It also shows that the 1kB mark is the crossover size for transitioning to a 3-Cycle access time.

Size	Lines	Access time (ns)	# of Clock Cycles
64B	2	0.55350625	2
128B	4	0.5580125	2
256B	8	0.567025	2
512B	16	0.58505	2
1kB	32	0.620896	2
2kB	64	0.64433	3

Table 5.4: CACTI Results

Based on these results it is prudent to investigate how the Indirection Cache advantage changes for Cache Access Times from 1 to 6 cycles, and to see when the breakeven point occurs (ie. the classical message copying method is faster than the Indirection Cache). Anything beyond 6 cycles becomes redundant for this investigation, as the Local Storage memory has an access time of 7 cycles [5]. Adding a separate memory entity to the architecture is only justifiable, if it improves upon the native memory access time. The following tables and figures show the results of this analysis. Examining the cache performance behavior beyond the estimated 2-3 cycles, to 6 cycles, will ensure that there is a margin of error and prove that the implementation is definitely beneficial.

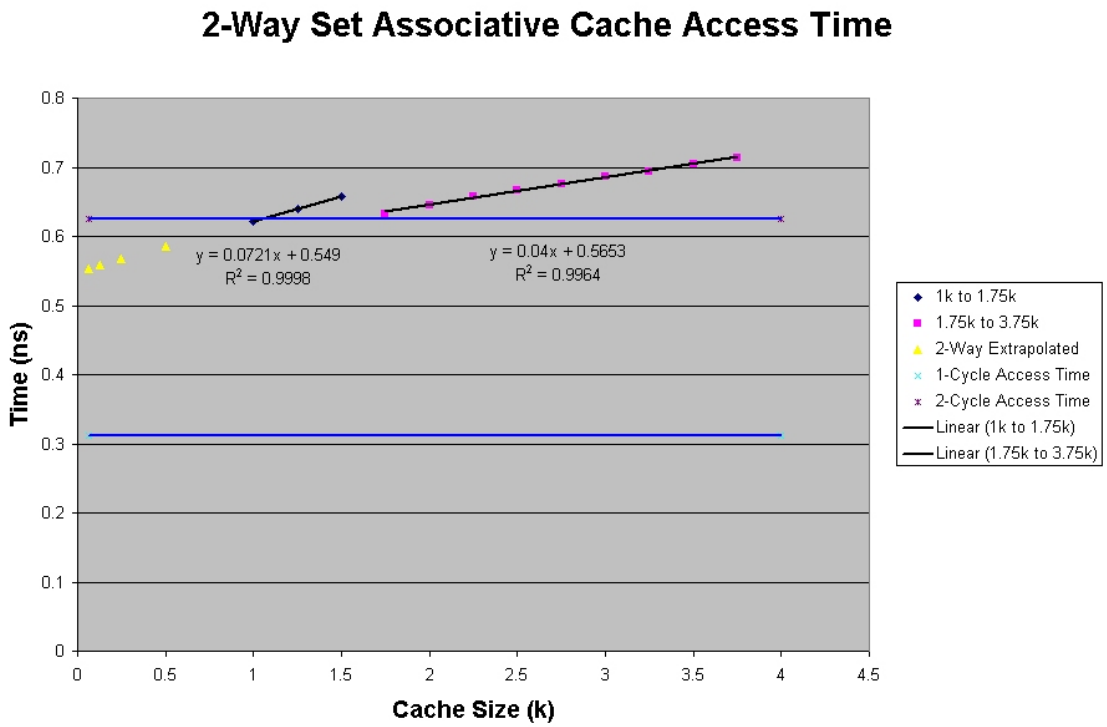

Figure 5.1: 2-Way Set Associative Cache Access Time
The two horizontal blue lines represent the threshold time for 1 and 2 clock cycles, of the Cell processor.
*results obtained using CACTI

5.2.1 Indirection Cache Improvement from Classical

This section compares the Indirection Cache (unbuffered) performance to the Classical transfer method for the range of cache cycle access times (1-6). First the 8/9 Processor Benchmark results are presented, then the 64 Processor Benchmark results are presented, and the section wraps up with an analysis of these results.

Benchmark	1 - Cycle	2 - Cycle	3 - Cycle	4 - Cycle	5 - Cycle	6 - Cycle
BT	4.17	2.09	1.39	1.04	0.83	0.70
CG	7.08	3.54	2.36	1.77	1.42	1.18
PSTSWM	3.19	1.59	1.06	0.80	0.64	0.53

Table 5.5: 8/9 Processor Indirection Cache Improvement from Classical

*An improvement multiplier less than 1 indicates a slowdown

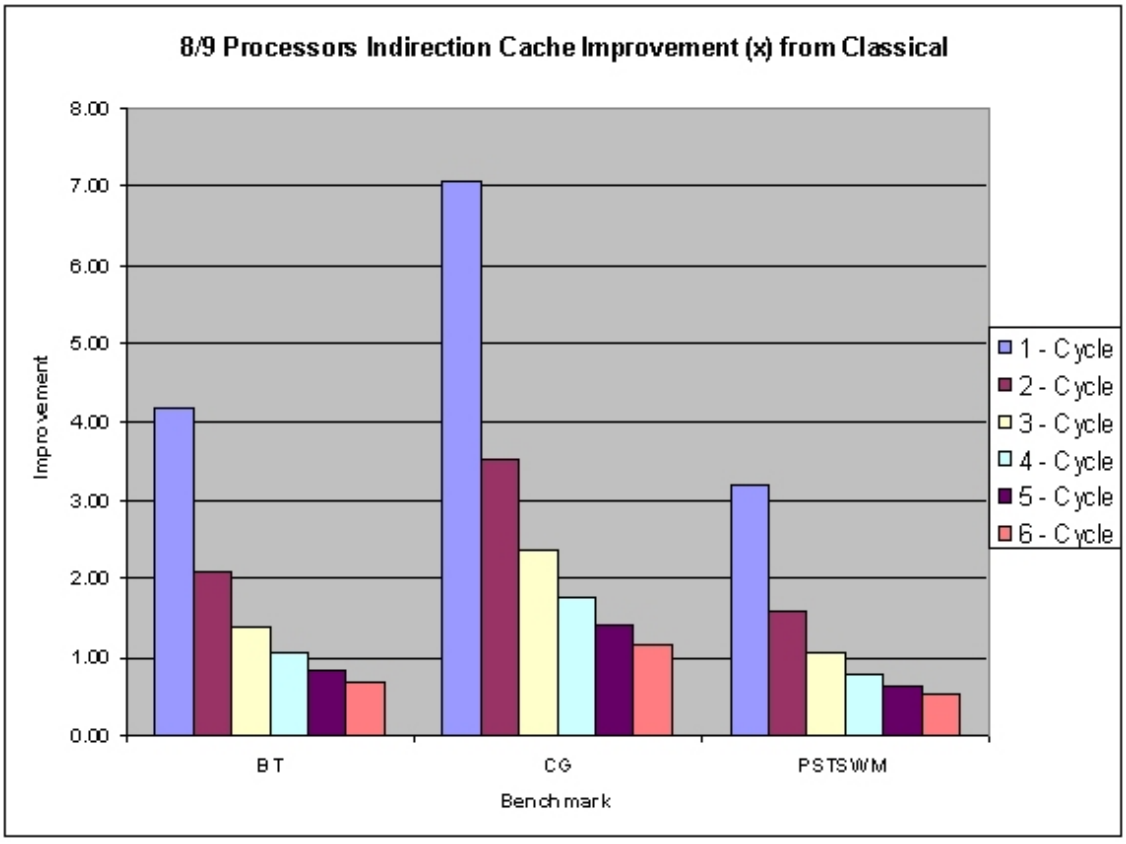

Figure 5.2: 8/9 Processor Indirection Cache Improvement from Classical

Benchmark	1 - Cycle	2 - Cycle	3 - Cycle	4 - Cycle	5 - Cycle	6 - Cycle
BT	4.77	2.39	1.59	1.19	0.95	0.80
CG	9.48	4.74	3.16	2.37	1.90	1.58
PSTSWM	3.44	1.72	1.15	0.86	0.69	0.57

Table 5.6: 64 Processor Indirection Cache Improvement from Classical

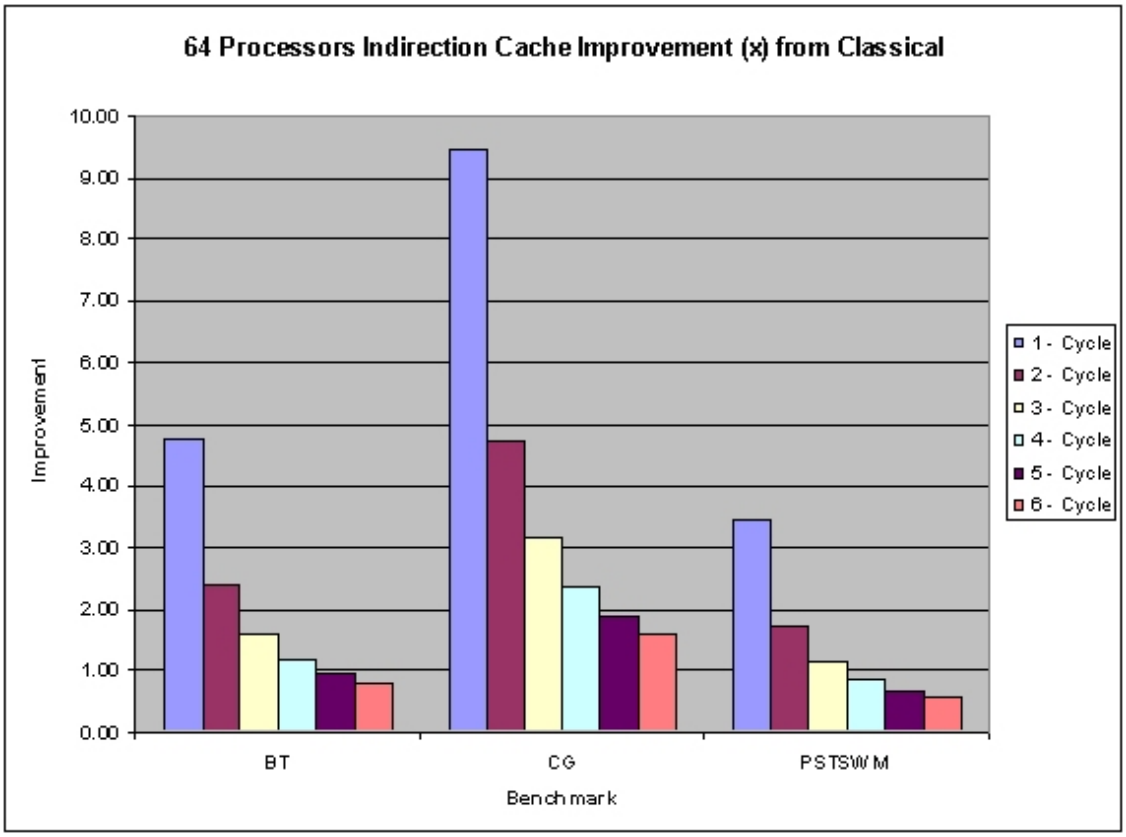

Figure 5.3: 64 Processor Indirection Cache Cycle & Classical Time Comparison

In general, the relationship between number of cycles and performance is that it degrades linearly, which is expected because the Indirection Cache performance is directly proportional to the cache access time. BT and PSTSWM show the lowest breakeven points for performance compared to the Classical method. For both BT runs, the breakeven point is between 4 and 5 cycles, and for both PSTSWM runs the breakeven point is between 3 and 4 cycles cache access time. This is encouraging to see, because it means that even the Indirection Cache without the additional buffers would still show improvement over the classical method. The next section examines the Benchmarks' improvement when an Indirection Buffer is used.

5.2.2 Buffered Indirection Cache Improvement from Classical

This section compares the Buffered Indirection Cache performance to the Classical transfer method for the range of cache cycle access times (1-6). First the 8/9 Processor Benchmark results are presented, then the 64 Processor Benchmark results are presented, and the section wraps up with an analysis of these results.

Benchmark	1 - Cycle	2 - Cycle	3 - Cycle	4 - Cycle	5 - Cycle	6 - Cycle
BT - 1	3781.48	1890.74	1260.49	945.37	756.30	630.25
BT - 2	4071.48	2035.74	1357.16	1017.87	814.30	678.58
CG - 1	1518.43	759.21	506.14	379.61	303.69	253.07
CG - 2	1532.68	766.34	510.89	383.17	306.54	255.45
PSTSWM - 1	163.34	81.67	54.45	40.84	32.67	27.22
PSTSWM - 2	164.59	82.29	54.86	41.15	32.92	27.43

Table 5.7: 8/9 Processor Indirection Buffer Improvement (x) from Classical

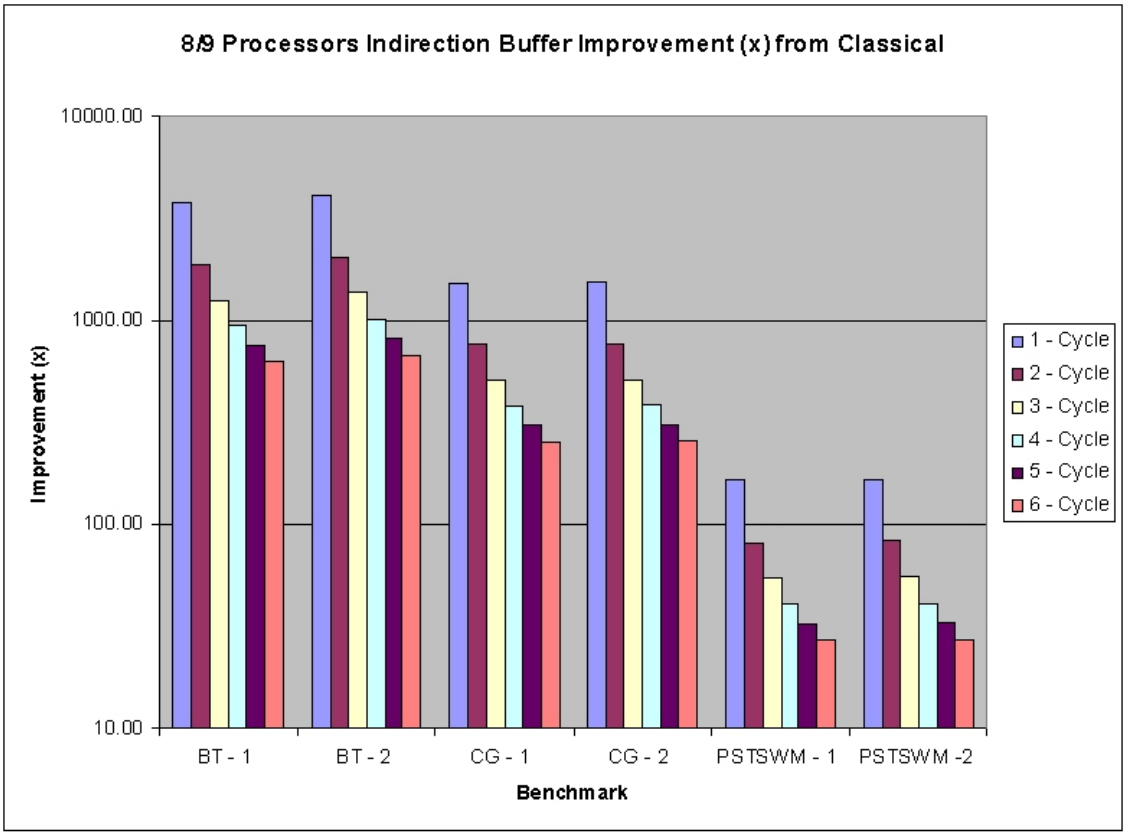

Figure 5.4: 8/9 Processors Improvement (x) from Classical (Logarithmic Scale)

Benchmark	1 - Cycle	2 - Cycle	3 - Cycle	4 - Cycle	5 - Cycle	6 - Cycle
BT - 1	708.32	354.16	236.11	177.08	141.66	118.05
BT - 2	812.44	406.22	270.81	203.11	162.49	135.41
CG - 1	910.41	455.21	303.47	227.60	182.08	151.74
CG - 2	1024.80	512.40	341.60	256.20	204.96	170.80
PSTSWM - 1	242.56	121.28	80.85	60.64	48.51	40.43
PSTSWM -2	248.13	124.07	82.71	62.03	49.63	41.36

Table 5.8: 64 Processor Indirection Buffer Improvement (x) from Classical (Logarithmic Scale)

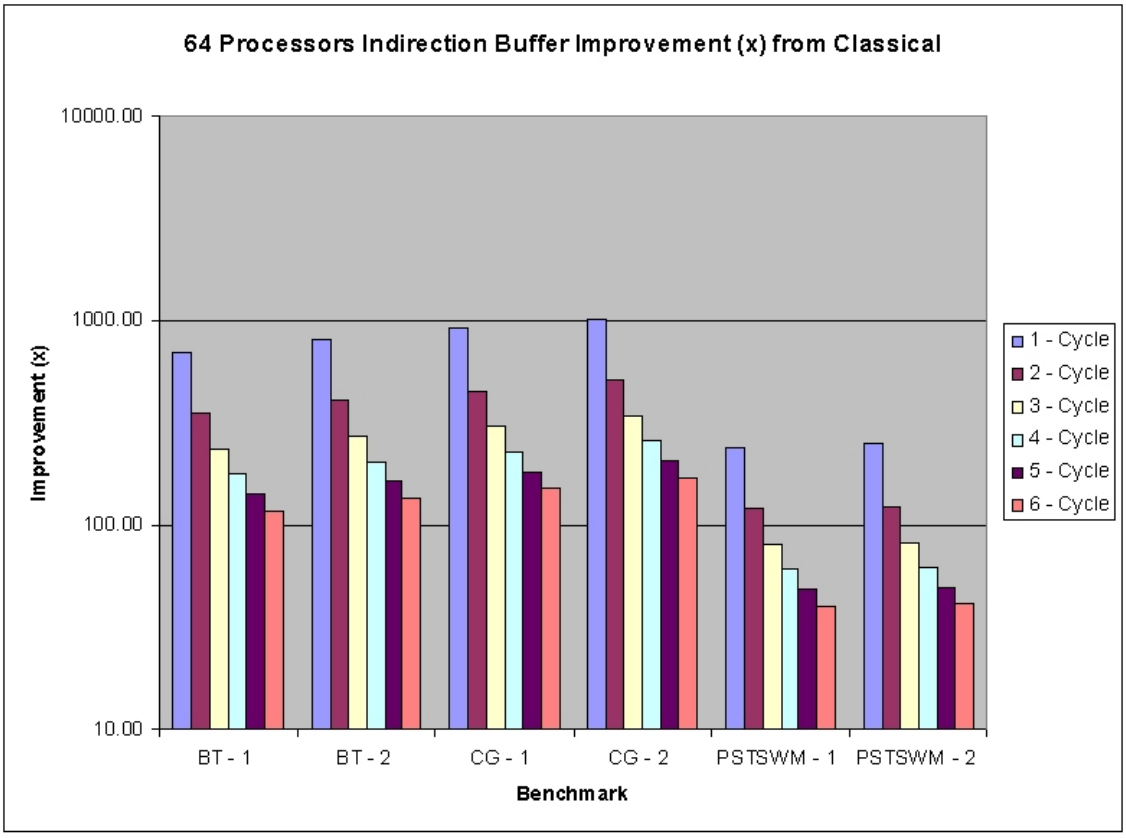

Figure 5.5: 64 Processors Improvement (x) from Classical

This approach shows that even in the worst case (6-Cycles, PSTSWM-9) there is a 27x improvement over the Classical Method. For the expected case of 3 cycles the best performance improvement is BT-9 at $\tilde{1}$000x, and the worst is PSTSWM-9 $\tilde{5}$0x improvement. Therefore, based on these benchmarks' results the Buffered Indirection Cache will greatly improve the message access time. Additionally, there is flexibility with the design, because even though the performance is sensitive to cache cycle access time it is not enough to negate the benefit.

5.3 Cache Energy Usage Minimization

The potential to reduce the energy usage of the Indirection Cache by only activating it when the Indirection Buffer is wrong, allows energy savings proportional to the Indirection Buffer hit-percentage (ie. 99% hits, then save 99% cache access energy). Tables 5.9 and 5.10 show the percentage of cache access energy saved by using this method. This analysis relies on the assumption that not only does the Indirection Buffer contain the destination address and valid bit, it also contains the receive variable address which can be checked quickly to confirm a correct prediction. This is equivalent to saying that the Indirection Buffer only holds the destination address and valid bit, and that there is an oracle present which correctly predicts validity 100% of the time (ie. the best case performance). As can be seen, because the Indirection Buffers have such a high hit percentage the savings are quite substantial. Therefore, when designing, the cache optimizations can be made for aggressive access times at the penalty of per-access enery usage (remembering that the total penalty over time is quite small because the cache only needs to be accessed on the Indirection Buffer misses).

Benchmark	Miss Percentage/Energy Savings
BT - 1	99.89
BT - 2	99.90
CG - 1	99.53
CG - 2	99.54
PSTSWM - 1	98.05
PSTSWM - 2	98.06

Table 5.9: 8/9 Processor Indirection Cache Access Energy Savings

Benchmark	Miss Percentage/Energy Savings
BT - 1	99.33
BT - 2	99.41
CG - 1	98.96
CG - 2	99.08
PSTSWM - 1	98.58
PSTSWM - 2	98.61

Table 5.10: 64 Processor Indirection Cache Access Energy Savings

Chapter 6

Epilogue

This Thesis extends the Indirection Cache concept introduced by Farshad Khunjush in his Thesis [13] and subsequent paper [12]. This Thesis accomplished this by optimizing the experimentation methodology, expanding the testing to a wider range of benchmarks (and sizes), and introducing a new architectural extension: the Indirection Buffer. It also investigated the design of the Indirection Cache, and how this effects the performance.

6.1 Conclusions

The basic Indirection Cache implementation shows slight improvement over the Classical Transfer Method, across all the Benchmarks that were examined. With just the basic Indirection Cache, there are some cases where individual receive variables have worse performance (ie. PSTSWM). Adding the Indirection Buffer mechanism gains at least 2 orders of magnitude time improvement over the regular Indirection Cache implementation. Examining how the Indirection Buffer performs as its access time increases (from 1-6 cycles), shows that the worst-case of 6 cycles, still is an order of magnitude (or 2) improvement over the Classical Transfer Method. Also, the Energy savings with a simple prediction and address-verification mechanism are proportional to the hit-rate of an Indirection Buffer, and therefore Energy usage can be kept to a minimum. These caching performance trends hold across all Benchmarks that were examined, and reinforce the superiority of the Indirection Buffer approach.

6.2 Future Work

In the immediate future, the next step would be to continue testing with other benchmarks that have different communication patterns (eg. SPEC MPI). The current Benchmarks and future ones should also be tested (if possible) with larger numbers of processors, to be as thorough as possible in observing the varying receive variable access patterns and sizes. Part of this study would involve looking more closely at the small variables, and the effect that the Indirection Buffer configuration has on their performance. Specifically, testing with more than two Indirection Buffer and also testing with different partitioning methods (for assigning variable addresses to specific buffers).

Another direction to investigate would be for the inclusion of very small variables (less than or equal to 8 Bytes), directly in the Indirection Cache. This is because under the current design assumptions there is room for a 64-bit address that points to their location in memory, which could instead be used to store their actual value. This would allow the cache to further improve access to smaller variables, which benefit less from the Indirection mechanism. In the same manner, it would also allow the Indirection buffer to hold the variables' value, eliminating all memory and cache accesses for cases where it predicts correctly. The overhead for this type of change would be minimal, as a single flag bit could be added to the design to mark the cases where this direct value access mechanism was in use.

It would also be advisable to migrate from the SimpleScalar simulator (which is slightly older), to a newer simulator and/or architecture. This effort would be quite involved however, because of the need to reinstrument and modify the new platform to accept the MPI Traces like SimpleScalar does currently. It is not recommended to use a multicore Simulator to simulate the MPI application directly, because the current maturity of the simulators forces the user to take a large simulation speed hit (ie. Most bottleneck at the points the simulation is forced to use a single core to simulate certain aspects of the multicore environment).

Appendix A

Logging Formats

This section includes a breakdown of the logging done through the benchmark instrumentation with PMPI. It will cover the data format used for the trace and payload files, and also which functions are instrumented to get this information.

A.1 Logging Formats

There are two files that are generated during the benchmark run, and then subsequently used to reconstruct the received messages within the simulator. The first file keeps track of all the necessary MPI_Recv function parameters (except the payload itself). Each line of this message trace file contains the following information:

```
fprintf(fp, "%s %d %d %d %d %d \n",
label, rankIn, source, count, datatype, tag);
```

The label corresponds to what operation is being logged (eg. MPI_Recv, MPI_IRecv, etc.). The rank is which node issued this command, and in the tests for this thesis is always zero. The source represents who the message is received from. Count corresponds to the number of elements the message will have. Datatype corresponds to what each element is (it is recorded as an integer because it uses the Fortran representation of the Datatype). The tag corresponds to the identifying tag passed along with the message. As can be seen, most of these fields correspond almost exactly with the arguments for the MPI_Recv and MPI_IRecv functions, which is where the logging occurs.

The second file logged is the payload file. This contains the message data that arrives at the receive variable address. The payload file is a binary file, and simply

contains the raw byte by byte data of each message written to the file, one after the other. To parse the file, it is assumed that the user has the accompanying trace file. In this way the first message will start at position 0 in the payload file and continue to position count*sizeof(Datatype) (ie. Byte length). The second message starts at this location, and it continues as far as its Byte length calculated from the trace file. Each subsequent messages payload data can be accessed in this way, repeating until the last message has been read in and the end of both files has been reached. This works well because during simulation the messages are read in order from the trace and payload files, as if they were being received normally.

A.2 Primary Functions Instrumented

The main functions that were instrumented were MPI_Recv, MPI_IRecv, MPI_Wait, and MPI_Waitall. The modified MPI_Recv starts by recording the trace file information, then makes the actual PMPI_Recv call. Because MPI_Recv is blocking, after PMPI_Recv has returned the message payload is available, so the payload data is logged to disk. For MPI_IRecv it starts by recording the trace file information, and then also queues a pending payload access, then calls PMPI_IRecv and returns (remember MPI_IRecv is nonblocking). The queue is necessary so that the payloads are only written to disk in the same order as the trace file. This is because multiple MPI_IRecvs can occur before calls to MPI_Wait or MPI_Waitall (which on their completion signal that the payload is available). In MPI_Wait the PMPI_Wait is called and when it returns the correct payload data is logged to disk. For MPI_Waitall the process is similar, but it iterates through outstanding requests in the queue order and writes each of their payloads to disk.

A.3 Exceptions and Additions

In addition to the receive operations being instrumented the MPI_Reduce and MPI_Allreduce operations are logged to separate files. In both of these cases the received data is recorded in the same format as the payload file (binary, one after the other). These operations are faked during simulation and the data is just read out of the files using fread(payloadfile,ByteLength), maintaining the file handle between accesses so that the next reduces data is read out each time. In the benchmarks tested, the reduce calls were made at the end of the simulation to get the final result or for timing.

MPI_Init and MPI_Finalize were instrumented too. MPI_Init was used to initialize some variables used in logging, and MPI_Finalize to dump some additional statistics at the end of the run. Both were used more during development and debugging.

Appendix B

Simulator Code

The SimpleScalar simulator had two additional source files added, to allow counting of variable accesses and MPI receive calls. The files are called mytable.h and mytable.c. The first section covers the macros and structures used by the additional code. The second section describes each of the functions, and how they are called from within SimpleScalar.

B.1 Primary Structs & Macros (mytable.h)

For the two indirection buffer solution proposed earlier, it was noted that the LSB of the address would be used to partition the accesses of the buffers. In other words, roughly half the receive variables would always map to the first buffer, and the other half to the second buffer. The exact hashing/mapping method is left for future study. For this initial investigation all that mattered, for each benchmark, was getting roughly half the accesses to occur on each buffer. Because the addresses assigned to each receiving variable change between benchmarks, a more dynamic approach was required to get the results for the most equal LSB partitioning. Therefore, the code was setup to test 8 different LSB partitions simultaneously. At least one of the 8 masks always creates a partition with 100% membership in one buffer, and zero in the other, because of the variable address alignment. Therefore the code is able to test for the single and double buffer scenarios at the same time. The following two macros facilitate this:

```
#define MASKCOUNT 8
#define USEFIRSTCACHE(X,Y) ( ((X)\&(0x1<<Y))==0 )
```

Additionally the structures for each receive variable being tracked kept 8 different switch counts. The same was also true for the total counts, which tracked 8 different sets of indirection buffers, and their corresponding switch (miss) and noSwitch (hit) counts.

```
struct mytable_elem
{
  md_addr_t DataAddress;
  int RcvCount;
  int DataSize;
  int Count;
  int switchCount[MASKCOUNT];
  int noSwitchCount[MASKCOUNT];
};

struct mytable
{
  struct mytable_elem Elements[20000];
  int TableSize;
  int switchCount[MASKCOUNT];
  int noSwitchCount[MASKCOUNT];
  int lastVarIndex[MASKCOUNT];

  int noSwitchCount2[MASKCOUNT];
  int switchCount2[MASKCOUNT];
  int lastVarIndex2[MASKCOUNT];
};
```

Note: One copy of the mytable struct is declared globally in SimpleScalar's sim-outorder.c function.

B.2 Added Functions

B.2.1 mytable_initialize

```
void mytable_initialize(struct mytable *table);
```

Called from sim_init(), which is called from main().
Order of operations:

- Logging files opened
- Table size set to 0
- Iterate through all sub-elements to initialize all counts to zero
- Iterate through global tracking values, setting counts to zero and initial indirection buffers to -1 (ie. no valid address to start).

Source:

```
void mytable_initialize(struct mytable *table)
{
  int i,j;
  logFileInit();
  table -> TableSize = 0;
  for ( i = 0; i < 20000; i++)
  {
table -> Elements[i].Count = 0;
table -> Elements[i].RcvCount = 0;
for( j=0;j<MASKCOUNT;j++)
{
table -> Elements[i].noSwitchCount[j] = 0;
table -> Elements[i].switchCount[j] = 0;
}
  }

  for( j=0;j<MASKCOUNT;j++)
  {
table->lastVarIndex[j] = -1;
table->switchCount[j] = 0;
table->noSwitchCount[j] = 0;

table->lastVarIndex2[j] = -1;
table->switchCount2[j] = 0;
```

```
table->noSwitchCount2[j] = 0;
  }
}
```

B.2.2 mytable_checkelem

```
int mytable_checkelem(struct mytable *table,
md_addr_t address);
```

Called from ruu_dispatch(), which is called from sim_main(), which is called from main(). More specifically, this function is called on a cache access to determine if the specified address is already being tracked in the table.
Order of operations:

- Iterate through list of elements, for each element find the start and end addresses and see if the passed in address falls within this range.
  - If a match is found, return the element array index
  - If no match is found, return -1

  This information is used to determine which function the simulator should call next. If it returns a matched index, then the counts and access information for that element should be incremented (mytable_increment). If it returns no match, then a new entry should be inserted (mytable_insert).

This function is also called in other locations which need to determine if the address being accessed is one of the receive variables being tracked. In these other cases, mytable_increment is called on a hit, but otherwise no action is taken (because it is for an address that is not a receive variable). Source:

```
int mytable_checkelem(struct mytable *table,
md_addr_t address)
{
  md_addr_t StartAddress, EndAddress;
  int i;
  int TableSize;
  TableSize = table -> TableSize;
  for ( i = 0; i < TableSize; i++){
```

```
    StartAddress = table -> Elements[i].DataAddress;
    EndAddress = StartAddress + table -> Elements[i].DataSize;
    if (address >= StartAddress && address <= EndAddress)
      return i;
  }
  return -1;
}
```

B.2.3 mytable_insert

```
int mytable_insert(struct mytable *table,
md_addr_t address,
int datasize);
```

Called from ruu_dispatch(), which is called from sim_main(), which is called from main(). More specifically, the function is called during a cache access, after mytable_checkelem has returned -1. This means that the cache access is the first time this receive variable has been seen (and because of the previous checks in the code, it is known to be a valid receive variable address that needs to be tracked.
Order of operations:

- Log that this is a new receive variable
- Insert into next free slot in the elements array
  - Set the address and datasize of this element
- Increment the size of the table
- Return the position of this new entry

Source:

```
int mytable_insert (struct mytable *table,
md_addr_t address,
int datasize){

  int TableSize;
```

```
  //Logging code for insert-starts
  logFileStart(address,datasize);

  TableSize = table -> TableSize;
  table -> Elements[TableSize].DataAddress = address;
  table -> Elements[TableSize].DataSize = datasize;
  table -> TableSize++;

  if (table -> TableSize == 20000)
    return -1;
  else
    return ((table -> TableSize) - 1);
}
```

B.2.4 mytable_increment

```
void mytable_increment(struct mytable *table,
int location,
md_addr_t address);
```

Is called from cache_access(), which is called from dcache_acces_fn(), which is called from many locations within the simulator. A call to mytable_checkelem() precedes this call, and only variables that fall within one of the receive variable address ranges are counted.
Order of Operations:

- Increment the count for the element in question
- Log this access
- Iterate through all masks, to determine Indirection Buffers behaviour, increment switches and noswitches as necessary

Source:

```
void mytable_increment(struct mytable *table,
int index,
md_addr_t address)
```

```
{
  int i;
  (table->Elements[index].Count)++ ;
  logFileAccess('i',address);
  //Test several different buffering mask scenarios all at once.
  for(i=0;i<MASKCOUNT;i++)
  {
  //Do this for each of the possible MASKS.
  //choose which 'cache' we might be using
  if(USEFIRSTCACHE(table -> Elements[index].DataAddress,i))
  {
  if(index == table->lastVarIndex[i])
  {
  //do nothing, this is not penalized
  table->noSwitchCount[i]++;
  table -> Elements[index].noSwitchCount[i]++;
  }else
  {
  //miss cache penalty, set new address index and increment count
  table->lastVarIndex[i] = index;
  table->switchCount[i]++;
  table -> Elements[index].switchCount[i]++;
  }
  }else
  {
  if(index == table->lastVarIndex2[i])
  {
  //do nothing, this is not penalized
  table->noSwitchCount2[i]++;
  table -> Elements[index].noSwitchCount[i]++;
  }else
  {
  //miss cache penalty, set new address index and increment count
  table->lastVarIndex2[i] = index;
  table->switchCount2[i]++;
```

```
  table -> Elements[index].switchCount[i]++;
     }
  }
  }//end loop
}
```

B.2.5 mytable_incRcvCount

```
void mytable_incRcvCount(struct mytable *table,
int location,
md_addr_t address);
```

Called from ruu_dispatch(), which is called from sim_main(), which is called from main(). Again, because of previous checks in the SimpleScalar code this is known to be a MPI_Recv call, and the index is known from the call to Mytable_checkelem(). Order of Operations:

- Increment the receive count for the element in question
- Log this access
- Iterate through all masks, to determine Indirection Buffers behaviour, and reset the indirection buffers as necessary (if the receive address might point to an old destination address).

Source:

```
void mytable_incRcvCount(struct mytable *table,
int index,
md_addr_t address)
{
int i;
(table->Elements[index].RcvCount)++ ;
logFileAccess('r',address);
//Test several different buffering mask scenarios all at once.
for(i=0;i<MASKCOUNT;i++)
{
  if(USEFIRSTCACHE(table -> Elements[index].DataAddress,i))
```

```
  {
  //new values have been loaded in to a variable.
  //do not increment a miss,
  //this will happen automatically on the next access.
  if(table->lastVarIndex[i] == index)
  {
  //invalidate this entry for the variable,
  //only if it was the one in there before
  table->lastVarIndex[i] = -1;
  }
  }else
  {
  if(table->lastVarIndex2[i] == index)
  {
  //invalidate this entry for the variable,
  //only if it was the one in there before
  table->lastVarIndex2[i] = -1;
  }
  }
}//end loop
}
```

B.2.6 mytable_printf

```
void mytable_printf(struct mytable *table);
```

This function is called from main(), when the simulation is complete so that the collected results can be dumped for the user to examine.
Order of operation:

- Iterate through each of the 8 masks and dump the column headings
- Iterate through each of the receive variables being tracked, and dump the collected counts (overall, and also the 8 indirection buffer switch/noswitch values).
- Iterate through each of the 8 masks for the total values, and dump this information at the end.

Source:

```
void mytable_printf(struct mytable *table)
{

  int i,j;

  printf("About to print mytable.\n");
  tablefp = (FILE *) fopen("mytableout.txt","w");
  fprintf(tablefp,"Address\t\tSize\tCount\tRcvCount");
  for(j=0;j<MASKCOUNT;j++)
  {
  //column headings.
  fprintf(tablefp,"\t|%d(Cache) \tSwchCnt\tNoSwchCnt ",j);
  }
  fprintf(tablefp,"\n");
  for ( i = 0; i < table->TableSize; i++)
  {
fprintf(tablefp,"0%08x\t%06d\t%07d\t%07d ",
table -> Elements[i].DataAddress,
table -> Elements[i].DataSize,
table -> Elements[i].Count,
table -> Elements[i].RcvCount );

  for(j=0;j<MASKCOUNT;j++)
  {
fprintf(tablefp,"|\t(%07d)\t%07d\t%07d",
USEFIRSTCACHE(table -> Elements[i].DataAddress,j),
table->Elements[i].switchCount[j],
table->Elements[i].noSwitchCount[j]);
  }
  fprintf(tablefp,"\n");
  }

fprintf(tablefp,"\n\nSwitch Count\t NoSwitch Count\t
Switch Count2\t NoSwitch Count2\n");
```

```
  for(j=0;j<MASKCOUNT;j++)
  {
fprintf(tablefp, "%07d \t\t %07d ",
table->switchCount[j],
table->noSwitchCount[j]);
fprintf(tablefp, "\t\t %07d \t\t %07d\n",
table->switchCount2[j],
table->noSwitchCount2[j]);
  }

  fclose(tablefp);
}
```

Bibliography

[1] A. Afsahi and N.J. Dimopoulos. Architectural extensions to support efficient communication using message prediction. In *Proceedings of the 16th Annual International Symposium on High Performance Computing Systems and Applications, HPCS*, pages 20–27, 06 2002.

[2] I. T. Association. Infiniband architecture specification. Available: AT http://www.infinibandta.org.

[3] T. Austin, E. Larson, and D. Ernst. Simplescalar: an infrastructure for computer system modeling. *IEEE Computer*, 35:59–67, 02 2002.

[4] D. H. Bailey, T. Harsis, W. Saphir, R. V. derWijngaart, A. Woo, and M. Yarrow. *The NAS Parallel Benchmarks 2.0: Report NAS-95-020*, 12 1995.

[5] C. Benthin, I. Wald, M. Scherbaum, and H. Friedrich. Ray tracing on the cell processor. *Interactive Ray Tracing 2006, IEEE Symposium on*, pages 15–23, 09 2006.

[6] F. Duarte and S. Wong. Cache-based memory copy hardware accelerator for multicore systems. *Computers, IEEE Transactions on*, 59:1494–1507, 11 2010.

[7] N. P. Jouppi et al. Architecting efficient interconnects for large caches with cacti 6.0. *IEEE MICRO*, 28.1:69–79, 2008.

[8] W. K. Giloi. Parallel supercomputer architectures and their programming models. *Parallel Computing*, 20:1443–1470, 1994.

[9] Michael Gschwind. The cell broadband engine: Exploiting multiple levels of parallelism in a chip multiprocessor. *International journal of parallel programming*, 35.3:233–262, 2007.

[10] G. Hager and G. Wellein. Architecture and performance characteristics of modern high performance computers. *Computational Many-Particle Physics*, 739:681–730, 2008.

[11] R. Huggahalli, R. Iyer, and S. Tetrick. Direct cache access for high bandwidth network i/o. In *Proceedings of ISCA-32, Madison, USA*, pages 50–59, 06 2005.

[12] F. Khunjush and N. J. Dimopoulos. Architectural enhancement for minimizing message delivery latency on cache-less architectures (e.g., cell be). In *Workshop on The Influence of I/O on Microprocessor Architecture (IOM-2009)*, 02 2009.

[13] Farshad Khunjush. *Architectural Enhancement for Message Passing Interconnects*. PhD thesis, University of Victoria, 2008.

[14] E. Leon, R. Riesen, K. Ferreira, and A. Maccabe. Cache injection for parallel applications. *International ACM Symposium on High-Performance Parallel and Distributed Computing*, 06 2011.

[15] E. A. Leon, K. B. Ferreira, and A. B. Maccabe. Reducing the impact of memory wall for i/o using cache injection. In *Proceedings of the 15th Annual IEEE Symposium on High-Performance Interconnects (Hot Interconnects), California*, pages 22–24, 08 2007.

[16] J. Liu, J. Wu, and D. Panda. High performance rdma-based mpi implementation over infiniband. *International Journal of Parallel Programming*, 32:167–198, 2003.

[17] Los alamos lab: High-performance computing: Roadrunner. Website, 11 2011. http://www.lanl.gov/roadrunner/.

[18] Message passing interface forum: Mpi 2.1. http://www.mpi-forum.org/.

[19] Mpi_recv. Website, 08 2004. http://www.mcs.anl.gov/research/projects/mpi/www/www3/MPI_Recv.html.

[20] Mpi_send. Website, 08 2004. http://www.mcs.anl.gov/research/projects/mpi/www/www3/MPI_Send.html.

[21] The message passing interface (mpi) standard. Website, 03 2009. http://www.mcs.anl.gov/research/projects/mpi/.

[22] Uvic rcf: Architecture. Website, 08 2011. http://rcf.uvic.ca/index.php?id=50.

[23] Uvic rcf: Minerva. Website, 08 2011. http://rcf.uvic.ca/index.php?id=7.

[24] What is pmpi? Website, 08 2011. http://www.open-mpi.org/faq/?category=perftools#PMPI.

[25] P. H. Worley and I. T. Foster. Parallel spectral transform shallow water model: A runtime-tunable parallel benchmark code. In *Proceedings of the Scalable High Performance Computing Conference*, pages 207–214, 1994.