Implementation and Analysis of Helper Threads with SSMT

Kurniadi Asrigo
EECG Department
993492617
kuas@eecg.toronto.edu

Chris Comis
EECG Department
993491636
comis@eecg.toronto.edu

Hassan Shojania
EECG Department
991257082
h.shojania@utoronto.ca

ABSTRACT

In this paper, several scenarios for helper threading were implemented and analyzed on the SPEC2000 mcf benchmark [1]. After determining the delinquent load of the mcf benchmark, three helper threading scenarios were implemented to prefetch the delinquent load into L2 cache. First a static loop-based helper process was implemented, followed by a static loop-based helper thread. Finally, a static sample-based helper threading was performed. All helper threads were simulated and evaluated on the SSMT simulator [2]. Our results indicate that sample-based triggering of helper threads results in the greatest improvement on the delinquent load, with a 71% increase in L2 hits and a 66% decrease in L2 misses, when compared to the single-threaded benchmark. However, sample-based triggering is very costly to the main thread as a whole. Consequently, loop-based triggering of the helper thread performs better on the benchmark as a whole, with only a mild speedup of 1.14%.

KEYWORDS

Delinquent Load, Simultaneous Multithreading, Helper Thread, SimpleScalar, SSMT, Static Triggering

1. INTRODUCTION

Despite quickly approaching physical limitations, processors continue to be developed at increased clock rates. Although clock rates continue to increase, memory access rates do not increase at the same rate. Hence, a speed gap is increasing between the processor clock rate and the latency of a memory miss. Consequently, the processor is spending significantly more time on memory stalls, waiting for the arrival of cache blocks [3].

To improve instruction-level parallelism (ILP), mid-90's processors incorporated the idea of superscalar processing [4], where multiple instructions may be executed concurrently on several different processing resources. By allowing out-of-order processing, a fortunate side effect of this technology is to mask the above-mentioned speed gap. Allowing instructions to be executed out of order, the processor may continue to process other instructions while memory misses are being serviced. Hence, as long as a memory misses is not in the critical execution path, its latency is no longer detrimental to overall program execution.

A natural extension to superscalar processing, which further improves instruction-level parallelism, is to additionally perform simultaneous multithreaded processing (SMT) [5]. Rather than allowing parallelism only at the instruction level, SMT attempts to find parallelism at the thread level. Granted that the speed gap continues to increase, it is expected that researchers now target methods to mask the latency of memory misses at the thread level.

Helper threading is one of the more promising techniques for masking cache miss latency at the thread level. In simplest terms, a single-threaded process is running on a multithreaded processor. In advance of a series of heavy loads (referred to hereon in as a delinquent load), the process spawns a helper thread and passes it all necessary live-in variables. Given all necessary information, the helper thread performs only the delinquent load operation. If the two threads are properly synchronized, the main thread will reach the delinquent load to find that all values have been pre-loaded into cache. As a result of helper threading, the research community expects that total execution time may be greatly decreased, because the negative performance impact of the delinquent load has been greatly reduced by the advanced work of the helper thread.

The purpose of this project is to use a simulator rather than real hardware to first isolate and determine the nature of delinquent loads in several single-threaded benchmarks. With this delinquent load information, we will then perform a variety of different helper threading scenarios. We will analyze the performance effects on the delinquent load, as well as the process as a whole.

The benchmarks that will be targeted include the Spec2000 mcf and art benchmarks, as well as the Spec95 go benchmark, all compiled to alpha-ecoff format. Simulations will be run on SSMT, which is an SMT extension to the SimpleScalar 3.0 out-of-order simulator [6]. The following describe each of the helper-threading scenarios that will be attempted.

Loop-based Triggering Using a Helper Process
As a first step, an independent process will be launched alongside the main process. Communication between the two processes will be performed using suspend/resume instructions and custom message-passing system calls. Furthermore, modifications must be made to the simulator to allow cache accesses by the helper context to be tagged as if they were loaded by the main context. At this point, synchronization will occur only at the start of the delinquent load.

Loop-based Triggering Using a Helper Thread
To build a more realistic helper threading scenario, a natural extension to the above is to create a helper thread from main.

Sharing the address space between the two threads has the following advantages:

- We will no longer need to modify the simulator to force helper cache accesses to appear as main cache accesses.
- Because the two threads share global variables, inter-process message passing is no longer necessary.

Sample-based Triggering Using a Helper Thread
To allow a more fine-grained synchronization between the two threads, the two threads may synchronize at several points inside the delinquent load loop.

2. BACKGROUND INFORMATION

The work described in this paper makes use of several tools, including the SimpleScalar out-of-order simulator, SSMT, an Alpha machine at the University of Toronto, and the Spec2000 benchmarks. Each one of these tools is briefly discussed below.

The SimpleScalar Out-of-Order Simulator
To analyze potential performance improvements in new processors architectures, an instruction-set simulator is necessary. Perhaps the most reputable simulator of this sort is the SimpleScalar toolset, a collection of Alpha and PISA ISA simulators, each with a different set of architectural features. At one end of the spectrum, the Simplescalar sim-fast simulator simulates only a basic processor architecture. At the other end of the spectrum, sim-out-of-order is, by far, the most complex of the SimpleScalar simulators. It simulates many advanced architectural features, including multilevel caches, speculative execution, branch prediction and instruction prefetching.

The simulator includes six pipeline stages, which must be walked through for each simulation cycle. These stages include:

- Fetch
- Dispatch
- Issue
- Refresh
- Writeback
- Commit

In actual simulation, the simulator performs the five stages in reverse order. Of important significance to the work presented in this paper, cache loads only occur in the issue phase and cache stores only occur in the commit phase.

SMT Simulation
Prior to commencing this work, we performed a survey of what SMT simulators are currently freely available. In the end, only two simulators were found. The first, SMTSIM [7], was written by Dean M. Tullsen. Following its one-time release in 1997 (), the simulator was met with mixed success. For example, the authors in [8] explain how SMTSIM 1.0 lacks the necessary primitive system calls to implement even the simplest Spec2000 benchmarks. However, the SMTSIM simulator was later extended to simulate the Intel Itanium ISA. Although this simulator is not freely available, its results are published in several papers [9][10].

The second simulator, SSMT, extends the SimpleScalar out-of-order simulator to perform simultaneous multithreading. This simulator is highly configurable, and there are also several primitive thread-level instructions in place. However, the SSMT simulator is not yet perfected, and several corrections had to be made to give the results presented in this paper.

There are two variations of SSMT simulator. One is based on SimpleScalar 3.0 and performs simulation of the Alpha ISA. The other is based on SimpleScalar 2.0 and supports PISA. We chose the Alpha version as it was using a more updated version of SimpleScalar. Hence, a DEC Alpha machine was necessary to compile the modified benchmarks, as well as the helper threads. Fortunately, the Canadian Institute for Theoretical Astrophysics (CITA) has a collection of powerful compute servers, including a Quad AlphaServer ES45 with 4x1GHz EV68 (21264) processors. We were fortunately able to get a temporary account on this server, and used this server for all benchmark and helper thread compilations.

Spec2000
To get any respectable recognition for our work, it was necessary that we select a respectable set of benchmarks. Hence, for the purposes of this project, the following Spec95 and Spec2000 benchmarks were used:

- go (Spec95)
- art (Spec2000)
- mcf (Spec2000)

Furthermore, because we are performing a very detailed instruction-level simulation, it was necessary that we choose minimal input vectors so that the simulations completed in a reasonable amount of time. Hence, for the go benchmark, the 2stone9.in input vector was used and the following command-line options were used to invoke the benchmark:

```
> go 50 9 2stone9.in
```

Unfortunately, for the art benchmark, we were not able to find many different input vectors. We therefore used the c756hel.in scanfile and the a10.img trainfile, as shown in the following command-line invocation:

```
> art –scanfile c756hel.in –trainfile1
a10.img –stride 2 –startx 130 –starty 220 –
endx 140 –endy 230 –objects 2
```

Finally, for the mcf benchmark, a small input vector (size of 35198 bytes) was found. The mcf benchmark was invoked as follows:

```
> mcf inp.in
```

3. RELATED WORK

Because of the aforementioned memory latency issues, the use of helper threads for prefetching of delinquent loads and branch prediction has become a very pressing research topic. Consequently, there have been a variety of different approaches for implementing helper threads, each with their own strengths and weaknesses.

In [3], the authors provide further analysis on helper threads. Unlike other work, the authors actually implement helper threading on an actual Pentium 4 processor. For loop-based triggering, the authors gain an average of 1.8% speedup over the entire program execution. For sample-based triggering, this value is increased to 5.5%. It is interesting to note that the authors do not select the same delinquent load as we did in our work. In fact, they discretely mention that mcf was the one benchmark for which they did not choose the most delinquent load for helper thread improvement. Instead, they sought improvement on two other mcf loads, the refresh_potential procedure and the price_out_impl procedure. After learning this, we looked at these loads in further detail. Although the loads suffered a modest amount of L1 misses, the number of L2 misses were well below those of the delinquent load (at approximately 40000 misses for refresh_potential and 150000 misses for price_out_impl). This is likely due to a difference in system configuration. Regardless, because of this, we did not look at these loads in further detail.

In [9], in addition to common methods of helper thread implementation such as loop-based and sample-based triggering, the authors propose a novel idea of "chained triggers." As opposed to only allowing a main thread to invoke a helper thread, the authors allow helper threads to invoke more helper threads. Chaining helper threads allows more parallel threads. On a SMT machine, this results in a higher throughput which may be advantageous for very heavy memory-access loads. Their results indicate that unlike a non-chaining helper thread system (which shows negligible improvement), a chaining helper thread system shows an average speedup of 76% over all benchmarks analyzed. In fact, the work presented in this paper is very similar to the work that we've discovered in that they use a similar set of benchmarks. Not surprisingly, when they examine the mcf benchmark in detail, the authors find an identical delinquent load to the one that we traced.

In [11], the authors provide a helper threading mechanism called data-driven multithreading (DDMT). Helper threads, referred to as data-driven threads (DDTs) are centered around the concept of register integration, where the main thread uses the direct results of DDTs via a shared register file. Although DDTs have the potential to drastically save data references in the main thread, the proposed mechanism is inflexible, in that it requires 100%-accurate DDT results. Hence, the ability to launch the speculative thread is clearly limited by the point at which we can guarantee DDT results will be 100% accurate. The authors achieve promising results. Using DDT helper threads that perform induction unrolling techniques, a maximum of 23.1% execution time for the mst benchmark, and 44.9% execution time for the em3d benchmark.

In [12], the authors introduce several new helper-thread mechanisms. Among these, the authors extend the work of [2] to propose a more flexible approach, which does not use register integration. Because the main thread does not directly incorporate the results of the helper threads, the results of the helper threads need not be 100% accurate. Because the results of the helper threads need not be 100% accurate, we are given more flexibility on when we may invoke the helper thread. The results reduce the number of cache misses by 64%.

In terms of experimental setup, the authors of [13] use an extended SimpleScalar simulator very similar to that used in our work. Rather than implementing helper threads, the authors of this paper propose and evaluate transparent threads, low-cost threads that are meant to be assigned to for low-priority tasks, and are intended not to interfere with the main thread. All authors are from the University of Maryland, and two out of the three authors contributed to changes and bug fixes in the SSMT simulator.

4. EXPERIMENT SETUP

As previously mentioned, the SSMT simulator was used in analyzing the performance effects of adding a helper thread. Most aspects of the SSMT simulator is highly configurable, and the following section describes the how the simulator was configured. Any modifications to this configuration will be discussed as they are made in subsequent sections.

According to [3], the Intel Pentium 4 processor has the following cache configuration:

L1 data cache	16KB
	4-way set associativity
	64 byte line-size
	LRU block replacement
L2 data cache	512KB
	8-way set associativity
	64 byte line-size
	LRU block replacement

We reconfigured the SSMT simulator to match the above Intel Pentium 4 cache configuration. Additional simulator configurations are summarized in Table 1.

Table 1 System Configuration

Configuration	Value
L1 data cache latency	1 cycle
L2 data cache latency	10 cycles
Memory latency	80 cycles[1]
Instruction fetch queue size	32
Instruction decode bandwidth	8
Instruction issue bandwidth	8
Instruction commit bandwidth	8
Register Update Unit size	128
Load/Store queue size	64
Branch Prediction	perfect
Execution order	out of order

With regards to simultaneous multithreaded processing, most resources were shared between contexts. The following summarize the resource sharing between contexts when simultaneous multithreaded processing was invoked:

- L1 and L2 data caches are shared between contexts
- At maximum, four threads could be fetched at each cycle
- branch predictor and BTB are shared
- 4 instruction blocks fetched for a context

5. PRELIMINARY BENCHMARK ANALYSIS

Before implementing any helper threads, the benchmarks had to be profiled. This involved modifying the simulator to record the cache statistics over the entire PC address space. This was a two step process. The first step involved profiling the entire address space of each benchmark to observe cache access statistics and

[1] Memory latency is 80 cycles, with an additional latency of 6 cycles per block.

potential delinquent loads. The second step involved detailed profiling of cache access statistics at the delinquent loads.

5.1 Determining the Location of Cache Access

Memory accesses happen in two stages of SimpleScalar, the issue stage for read accesses and the commit stage for write accesses. As all of our identified delinquent accesses happened to be load operations, we disabled profiling of store operations.

5.2 The Quest for Delinquent Loads

As a first step, the size of the executable PC address space had to be determined. This involved a simple simulation, where the highest and lowest PC values were recorded. These two values may be subtracted, and the result may to obtain the size of the PC address space. For each benchmark, the PC address space is summarized below.

go (spec95)	680844
art (spec2000)	387420
mcf (spec2000)	377532

Given the executed PC address space, additional code may be added at the issue and commit phases in the simulator to profile the cache statistics at each PC address.. The cache statistics recorded include L1 hits, L2 hits and L2 misses. The following table summarizes results for all three benchmarks, and the following figure shows L2 cache misses for the three benchmarks over their entire PC address spectrum. Note that for each figure, the start PC is the leftmost entry along the x-axis.

Table 2 Cache Profile Statistics

Benchmark	Delinquent PC (hex)	L1 hits	L2 hits	L2 misses
art	200100f4	190463	5896	42090000
go	20055778	1113605	96043	13594
mcf	2000f558	269	958105	1034440

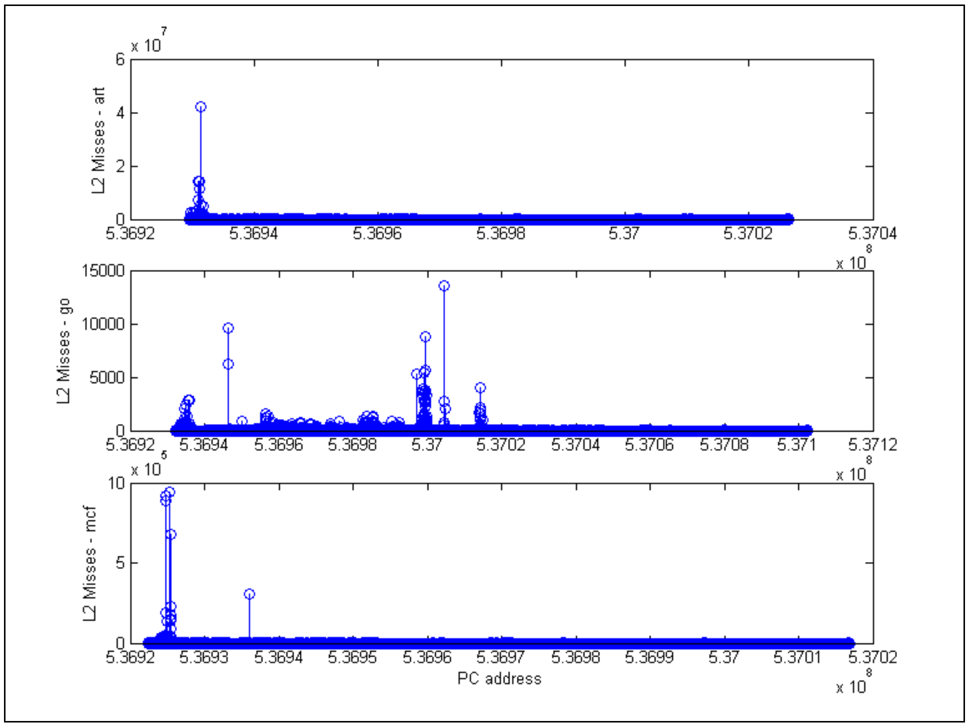


Figure 1 Profiled L2 Miss Statistics

From Figure 1, it may be seen that the go benchmark does not have a large delinquent load. Consequently, it will not be analyzed any further. The art benchmark has a very large delinquent load. However, the art benchmark, in its entirety, took approximately 13 hours to simulate. Given this long simulation time, it would be impossible to debug and run many different helper threading scenarios on the art benchmark. Hence, this benchmark was also eliminated. The mcf benchmark is well suited for improvement. It took approximately 40 minutes to complete, and had a significant delinquent load that could be improved.

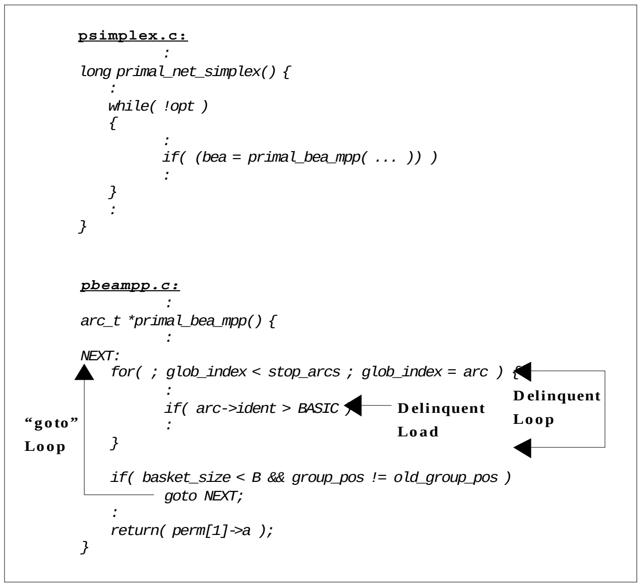
5.3 Detailed Delinquent Load Analysis

Because mcf was going to be our target benchmark, significant time was spent understanding the benchmark. First, the delinquent load was identified. Then, the benchmark was analyzed as a whole. This will have a direct impact on how we implement our helper threads, and how well the helper threads perform.

By analyzing further, the delinquent load was traced to a pointer access inside a nested for-loop, as shown in the following disassembly from the mcf pbeampp.c source file.

```
/* the nested for-loop begins here */
for( ; arc < stop_arcs; arc += nr_group )
0x2000f550:     cmpult  s1, s5, t12
0x2000f554:     beq  t12, 0x2000f614
{
   /* this if-statement contains the access */
   if( arc->ident > BASIC )
   /* this is the delinquent load */
   0x2000f558:   ldq  t0, 56(s1)
   0x2000f55c:   ldq_u  zero, 0(sp)
   0x2000f560:   ble  t0, 0x2000f600
   {
   /* the iterative for-loop continues */
```

To get more insight into how the delinquent load behaves, we took a closer look at the code surrounding the delinquent load, which is paraphrased in Figure 2.



```
psimplex.c:
        :
long primal_net_simplex() {
    :
    while( !opt )
    {
        :
        if( (bea = primal_bea_mpp( ... )) )
        :
    }
    :
}

pbeampp.c:
        :
arc_t *primal_bea_mpp() {
        :
NEXT:
    for( ; glob_index < stop_arcs ; glob_index = arc ) {
        :
        if( arc->ident > BASIC )
        :
    }

    if( basket_size < B && group_pos != old_group_pos )
        goto NEXT;
    :
    return( perm[1]->a );
}
```

Figure 2 Program Control Flow of mcf Benchmark

The following three points help explain the delinquent load access patterns.

- The delinquent load is being invoked through the whole program execution. Also, the following points give a more detailed description of the delinquent load, and how it iterates inside the benchmark. These points will be fundamental in determining how the helper thread seeks to improve the benchmark.
- The benchmark consists of two phases, denoted Phase 1 and Phase 2.
- We will refer to the innermost nested for-loop as the delinquent loop. This for-loop is iterated between 296 and 300 successive times.

Further analysis on the benchmark shows that there are significantly different cache access patterns between the first and second phases.

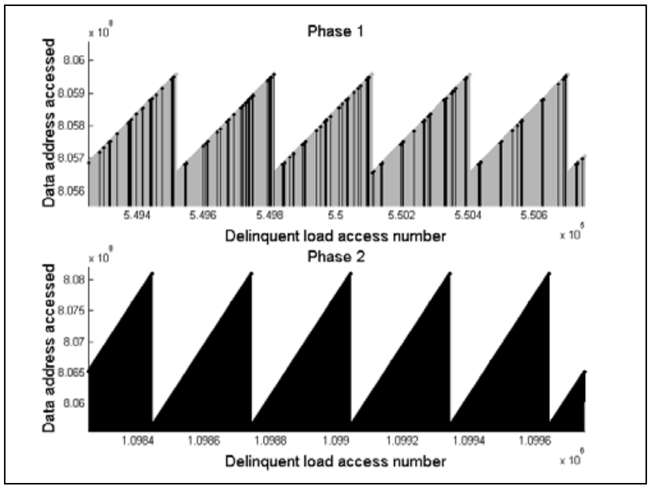


Figure 3 Detailed L2 Cache Profile Results - mcf
Light: L2 cache hits
Dark: L2 cache misses

Figure 2 shows approximately 5 iterations of the delinquent load (recall, each delinquent load is approximately 300 iterations). As seen in Figure 3, the second phase accounts for almost all of the L2 cache misses, while the first phase behaves relatively well. This data is quantified in Table 2.

Table 2 Cache Access Performance

Phase	L1 hits (%)	L2 hits (%)	L2 misses (%)
1	0.1	86.0	13.9
2	0	1.5	98.5

By looking further into the mcf code, we may determine these two phases are so different.

- In the first phase, the nr_group variable increments by 16. Hence, the address of arc in the delinquent load increments by 1024 (considering the structure size).
- In the second phase, the nr_group variable increments by 128. Hence, the address of arc in the delinquent load increments by 8096.

To explain why this has a large impact on cache performance, recall our current L2 cache configuration. Each cache block is 64 bytes in size. The cache is 8-way set associative, with a total of 1024 sets.

Phase 1 Analysis
In Phase 1, because nr_group is 16, two successive accesses are loaded into two sets that are 16 apart.

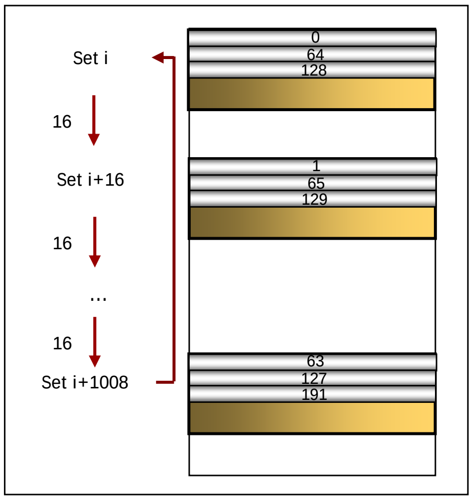


Figure 4 Cache Access Patterns for Phase 1

Because we have a total of 1024 L2 cache sets, it would take 1024/16 = 64 successive accesses before we access arrive at our original cache line. Because each cache line is 8-way associative, we would require 8*64 = 512 successive accesses before we start thrashing out previously loaded values. This initial analysis suggests that thrashing will not be a problem for the first phase of the benchmark.

Phase 2 Analysis
In Phase 2, because nr_group is 128, two successive accesses are loaded into two sets that are 128 apart.

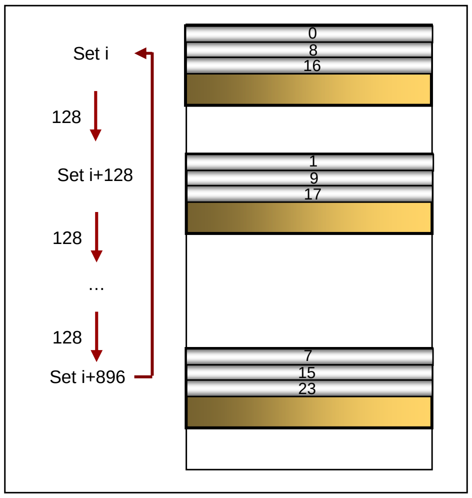


Figure 5 Cache Access Patterns for Phase 2

Because we have a total of 1024 L2 cache sets, it would take 1024/128 = 8 successive accesses before we access arrive at our original cache line. Because each cache line is 8-way associative, we would require 8*8 = 64 successive accesses before we start thrashing out previously loaded values. Furthermore, this analysis assumes that the delinquent loop is the only loop that may load into these sets. Clearly, this is not the case over the course of the entire delinquent loop. The above analysis suggests that thrashing may be a problem for the second phase if the helper thread becomes unsynchronized with the main thread (i.e. the prefetch is too far ahead). Hence, when implementing the helper thread, synchronization will play a factor in helper thread improvement.

6. HELPER IMPLEMENTATION

In all our implementations, we manually analyzed the delinquent loads, formed the p-slice (prefetch slice [3]) in the helper and added trigger/synchronization schemes between the two threads. Ideally this process can be implemented automatically: running profiler after compilation and using the captured profile data as a feedback into the compiler to automatically form p-slices, launch the helper thread with proper synchronization already embedded within main and helper thread [3].

6.1 Live-in variables

All the data structures and system information that is needed are calculated offline, for example the start of the iterations, the number of iterations and the end of the iterations. The main thread will pass the iteration information using *my_sd struct*, as shown below:

```
my_sd.start = initialize ? arcs : (arcs+group_pos) ;
temp = (stop_arcs-my_sd.start);
my_sd.it_advance = initialize ? (((m-1) / K ) + 1) : nr_group;
my_sd.it_num = ((temp%my_sd.it_advance)==0) ? (temp/my_sd.it_advance) :
                        ((temp/my_sd.it_advance)+1) ;
```

We set the helper thread to detect the second phase of mcf so it stops at 64 iterations to prevent trashing. However in the future we implement the helper thread monitoring of the main thread stage using *glob_index* as below:

main thread delinquent load:
```
glob_index = arc;
for( ; glob_index < stop_arcs ; glob_index = arc ) {
                :
                arc += nr_group;
}
```

6.2 Helper Process with Loop-based Synchronization

In the beginning of the project, because of lack of documentation, we found launching multiple processes much easier than multiple threads. So we employed a prefetching helper process first instead of a helper thread. This helper process started on a separate context (i.e. One of the multiple contexts supported by the simulated SMT processor) right from the beginning. We made some modifications to the process activation of SSMT as described in Appendix 1.

SSMT's main contribution is the addition of a whole set of context related instructions (e.g. to suspend a context, resume it, pass messages between different contexts). These context instructions can be used quite transparently no matter if the contexts are launched as a process or a thread. But communicating data between two processes require *Inter-Process Communication* mechanism which is not yet supported in the SSMT. We implemented two new system calls to pass data from one process to another process. This was done through a mailbox fashion where *SetPrivateData* leaves the data in the receiver's mailbox and a subsequent *GetPrivateData* picks the data. This data is temporarily kept by the simulator as it emulates the role of the operating system. Since our communication scheme designed very simple (unidirectional communication from main to helper process where the communication points already synchronized through context instructions), the implementation of such mechanism was not complex.

After implementing our helper process mechanism, it was found that all cache accesses of the main thread were missing even though they were just recently prefetched by the helper process. This was surprising as the hardware was configured with shared caches instead of separate per context caches. This was apparently because of cache tagging mechanism of SSMT where virtual address tagging was used instead of physical address tagging. Each cache line was tagged with the context ID of owner's context preventing data sharing between contexts (basically only cache capacity was shared). We had to make a hack to fix this as described in more detail in Appendix 1.

Scenario

We will now briefly describe the helper process scenario. The main and helper process are started as separate process by the simulator (see Figure 6) (1). The helper thread then will suspends itself (2) while main will execute until it reach the point (3) where the live-in variables of delinquent load can be determined and it is safe to resume the helper process to the load the delinquent load into the cache.

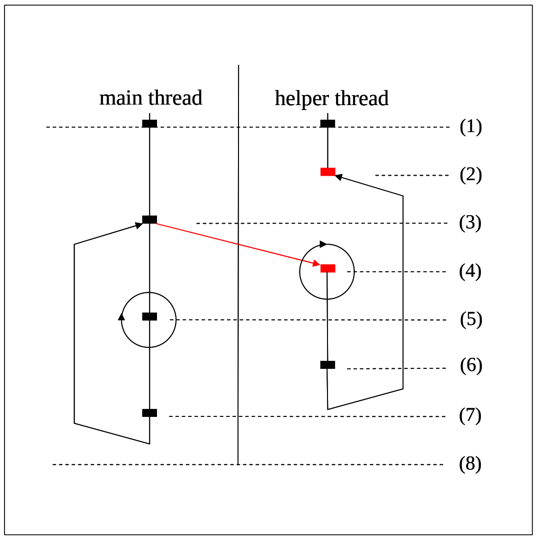


Figure 6 Control Flow Diagram for Loop-Based Triggering

At that point, main will execute the embedded code that makes the *SetPrivateDataSystemCall* to send the delinquent load information to helper The main process after sending the data it will make LCU context instruction (4) to wake up the helper process that is currently suspending. Then it will continue its normal execution into the delinquent load (5).

The helper after it wakes up, it will first call *GetPrivateDataSystemCall* to get the information passed by main process then execute its delinquent loop (4). Once the helper process finished (6) its delinquent loop, it will go back and suspend itself waiting for the next request.

At the end of the main program, it executes another embedded code to send signal to the helper process to stop and exit.

Problem with this approach

We realized the nature of MCF has 2 phases of execution. We were able to help the first phase but the helper thread ran too far ahead for the second phase that it actually thrashed its own memory access before the main actually get the data from the cache.

Because of the lack of synchronization mechanisms that can be done between 2 processes, it would make the solution be very complex and unnatural. It would not make more sense to continue this approach further. However in this approach, we achieved modest results by limiting the number of iterations to 64.

6.3 Helper Thread with Loop-based Synchronization

The helper process method had its own limitation, mainly that inefficient communication between the main and helper resulted in the major problems of the prior approach which is the lack of synchronization between the two processes. This was a much more serious issue for the second phase as cache trashing can happen only after 64 iterations of the prefetch load (even in ideal condition). So, for the second phase, we had to limit the iterations to 64 only (around 20% of a full prefetch). Otherwise we needed to implement a re-trigger mechanism of another chunk of 64 iterations from within the loop containing delinquent loads. Since the loop containing the delinquent load was repeated around 2 million times within the program. Any extra code added there was seriously impeding the progress of main thread.

The other possible solution was to make helper process monitoring the progress of the main process as it proceeds with the delinquent loads and restart prefetch for another chunk of 64 iterations. But this monitoring would require continuous inter-process call which wasn't feasible. So we decided to implement the helper as a real helper thread to be able to access main thread's variables (specially the induction variable of delinquent loop) transparently.

After fixing the thread creation issues (see Appendix 1 for details), we implemented a similar prefetch algorithm for the helper thread as the one we already used in the helper process scenario (see Figure 6). Naturally we didn't expect to gain major performance improvements here as both approaches were doing the same thing but with different implementation. The only difference was that the inter-process communication system calls for communicating live-in variables were no longer necessary and a global data structure was enough to share data between the two threads.

The scenario

This scenario is very similar with the previous one. The differences are the start up and the synchronization mechanism as a result of the threading mechanism.

The whole program is started with only 1 context as one process. The helper thread is created using *createThread* mechanism that is embedded at the start of the main process. This new thread is set to run on different context but still uses the same memory address space. After it has started, the helper thread suspends itself using the LCU instructions mechanism.

The main thread reaches the earliest point where the live-in variable can be determined. It then checks if the helper thread is busy (notice we may perform a higher level of synchronization because we are using a threading mechanism rather than an individual process) before it sends the data to the helper thread using global variables. The subset of code can be seen at below. The program ended with the same mechanism as previous scenario.

The helper thread wakes up after the signal then sets itself to busy mode. Then read the global information passed by the main thread and executes the cache accesses. In this part, it will detect if it is in the second phase, it will stop at 64 iterations with the reason previously explains. The code subset below to better explains helper thread mechanism:

```
while (1) {
    HELPER_BUSY = 0;

    /* Wait for next request */
    SSMT_LCU(0, 2) ;

    /* Exit condition */
    if (my_sd.start == NULL)
        break;

    HELPER_BUSY = 1;
    :
    // Initialize the iterations parameters
    :
    for ( ... ) {
        // Do memory access of delinquent load
        :
    }
    :
}
```

6.4 Helper Thread with Sample-based Synchronization

As mentioned in the previous section, sample-based synchronization of prefetching with the main thread was necessary to prevent cache trashing beyond the first 64 iterations.

The Scenario

Most of the scenario here was similar to the previous case, where the simulator starts only 1 process (1) (See Figure 7) the main thread created the helper thread in the beginning (2) and the helper suspends itself right after startup (2). When the main thread enters *pbeampp*, it updates the shared live-in variable structure and signals the helper to start (3) as before. Then helper thread starts prefetching (4). As before, for phase1 the whole prefetching is done in one single shot (single prefetch loop) while for phase 2 one chunk of 64 iterations are prefetched. But now for phase 2 we start monitoring the progress of main thread's state in delinquent loop (7) and after making sure that enough that the prefetched data are consumed by the main thread then helper thread resumes to another chunk of 64 iterations (go back to 4). Effectively the full range of prefetch iteration is divided into several smaller iterations. This was to guarantee that the new chunk of iterations would not trash the previously prefetched chunk.

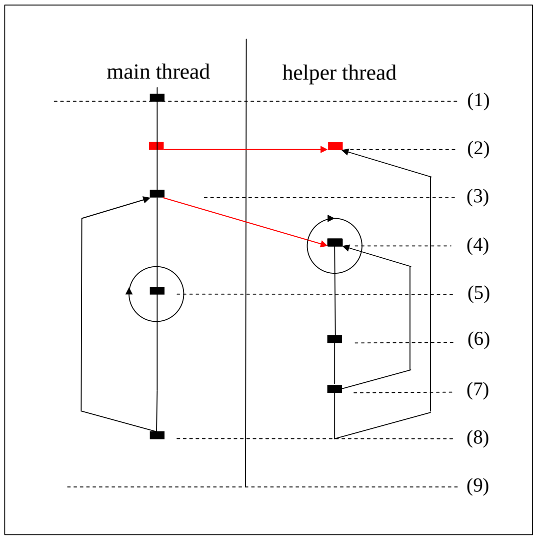


Figure 7 Control Flow Diagram for Sample-Based Triggering

This was achieved by two-way monitoring between the main and helper threads. In a more ideal solution, helper thread checks the induction variable of delinquent loop and backs off for a little while by sleeping. But a major problem facing us was that SimpleScalar implements system calls (including sleep) like a single instruction and simulates its effect on the host machine. So effectively a sleep call, sleeps the whole simulation not affecting the thread scheduling (i.e. fetching from the awake threads only) as SimpleScalar itself is not multi-threaded and only has one simulation thread.

So we had to resort to constant polling of main thread's induction variable (at 7) which resulted in huge increase of total executed instructions (e.g. From 270 million to 550 million). This had serious effect on the fetch bandwidth available to the main thread as helper thread was executing a long time and only going to suspension only when main thread finishes the whole delinquent loop (remember that frequent trigger of the helper by main thread was not a good option as explained in the previous section). An ideal solution was to come up with a special new instruction that suspends the helper thread for a *pre-determined* number of cycles allowing the main thread to have full access to the fetch bandwidth. Because of lack of time, we could not implement such an instruction so far.

7. RESULTS

The helper threads from Section 6 were implemented on the SSMT simulator. Results were obtained for many different configurations. For example, in the loop-based synchronization, the number of subsequent prefetches was modified between the ranges of 32 and 64. Furthermore, for the for the sample-based helper thread, we experimented with three different parameters:

- Chunk size - The number of successive iterations performed by the helper thread.
- Alpha: the number of iterations, in advance of the next delinquent loop, the main thread should trigger the helper thread.
- Beta: the number of iterations for the next delinquent loop that the helper thread should skip before beginning delinquent accesses.

Please refer to the code for more information on these different parameters. Due to space constraints, we will only discuss the most successful results.

7.1 Loop-Based Helper Process Triggering

Recall that for loop-based helper helping, the second phase has a problem with thrashing values if more than 64 successive accesses are performed before the main thread has consumed. Because of this, the helper process performs all loop iterations for the first phase of mcf. However, for the second phase, only the first 64 delinquent loop iterations are performed. Figure 8 shows the results of Phase 1, while Figure 9 shows the results of Phase 2. The darker regions of both figures correspond to L2 cache misses. Compared to the Figure 3, it can be seen that the helper thread has a modest improvement on phase 1 and a significant improvement on Phase 2. We will analyze phase 2 in results in more detail, as this phase has a much higher potential for improvement. For Phase 2, for the first 64 loop iterations, the helper thread mostly misses. This results in a decreased number of misses in the main process. Hence, using the correct synchronization mechanism, there is possibility to actually improve the main thread hit rate in the mcf Phase 2 region.

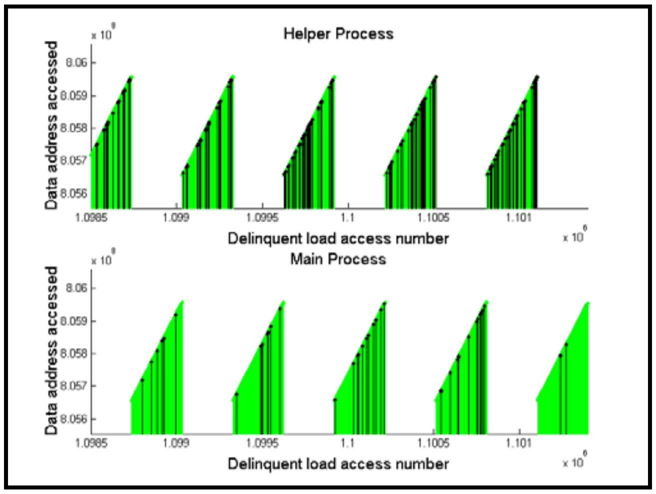


Figure 8 Loop-based Helper Process Phase 1 Results

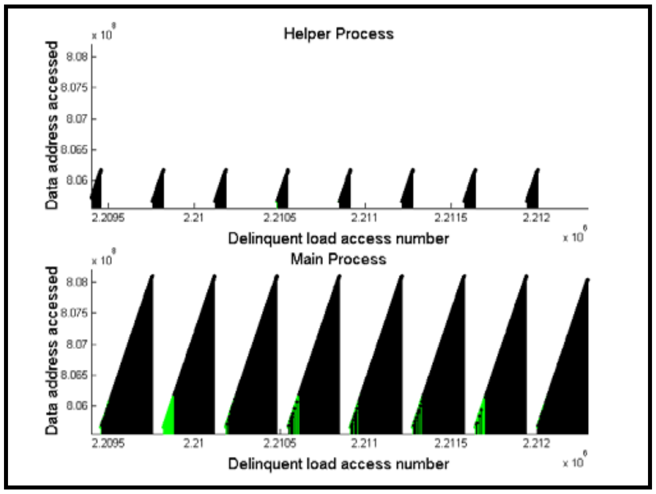


Figure 9 Loop-based Helper Process Phase 2 Results

Looking only at the delinquent load cache statistics, Table 3, shows that there are improvements in the cache hits the L2 cache for phase 2 and significant L2 cache improvements for phase 2 (see Table 2 for the original mcf performance metrics). There is a very small improvement in hits of L1 cache for phase 1. This occurs when the main process is running very close behind the helper process. In the case of L2 hits, Phase 1 improvements are significant and by performing a reduced number of accesses (64) for Phase 2, we have achieved 16% improvement in the delinquent load. The potential for thrashing in Phase 2 limits us to perform more prefetching from the helper process, and hence, limits potential performance improvements.

Table 3 mcf Performance with Loop-Based Helper Process

Phase	L1 hits (%)	L2 hits (%)	L2 misses (%)
	Helper		
1	0.07	81.75	18.18
2	0.98	0.36	98.66
	Main		
1	0.09	96.42	3.49
2	0.00	15.99	84.01

7.2 Loop-Based Helper Thread Triggering

Figure 10 below shows only the second phase results of the mcf benchmark. For this helper threading scenario, we treat the Phase 1 portion of the benchmark identical to the method discussed in the previous scenario. Hence, we will not discuss the Phase 1 results in detail. As before, the possibility of thrashing exists in Phase 2. As a result, for each delinquent loop, we were forced to stop helper thread execution after the first 64 memory accesses.

Once again, for Phase 2, we were able to achieve modest improvements from the helper process. Although the performance results are comparable to those in Section 7.1, the fact that we are now using a helper thread rather than a helper process allow us to explore more fine-grained and cleaner synchronization methods.

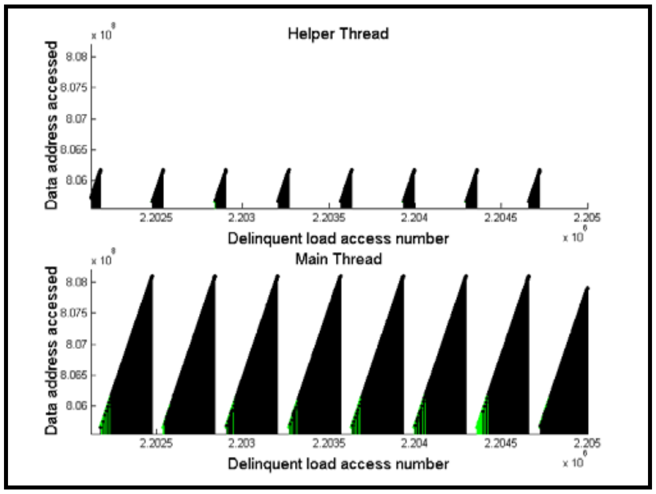


Figure 10 Loop-based Helper Thread Phase 2 Results

Table 4 below shows the results of this helper threading scenario. As can be seen, we see very similar improvement compared to the previous section.

Table 4 mcf Performance with Loop-Based Helper Thread

Phase	L1 hits (%)	L2 hits (%)	L2 misses (%)
	helper		
1	0.03	85.47	14.51
2	0.01	0.51	99.48
	main		
1	0.08	96.53	3.39
2	0.01	16.01	83.99

7.3 Sample-Based Helper Thread Triggering

For this implementation, Phase 1 implementation was again not changed. However, sample-based synchronization was performed for Phase 2. Many different configurations were attempted for static sample-based helper threads. From previous discussion (see Figure 4 and Figure 5), it was determined that, at maximum, 64 loop iterations may be performed by the helper thread before we begin thrashing out previously loaded values. Hence, this was the upper bound for the number of iterations performed by the helper thread. In addition to this parameter, referred to as the chunk size, there were two other parameters that were experimented with.

For the previously-mentioned parameters, the chunk size was varied from 32 to 64. The alpha parameter was varied from 0 to 10 and the beta parameter was varied from 0 to 10. The following figure represents the best possible configuration, where the chunk size was 32, alpha was set to 5 and beta was set to 0. As can be seen, the helper thread monitors the main thread. When a certain iteration is reached, the helper thread begins delinquent load accesses. This occurs at several points inside the delinquent loop and it can be seen that there is actually overlap between the start of main thread and the end of helper thread execution.

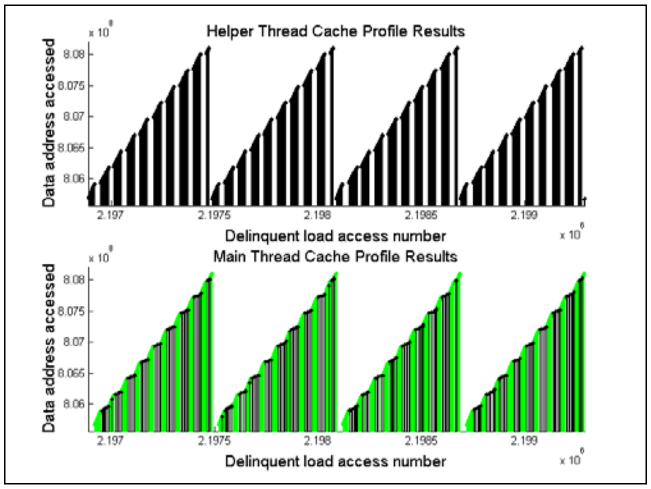


Figure 11 Sample-based Helper Thread Phase 2 Results

Table 5 shows the improvements for this method. By looking at the Phase 2 results, main thread L2 Cache misses were significantly decreased, and hits improved significantly from 1.5% in the original mcf (Table 2) to 66%.

Table 5 mcf Performance with Sample-Based Helper Thread

Phase	L1 hits (%)	L2 hits (%)	L2 misses (%)
	helper		
1	0.05	91.76	8.19
2	0.00	0.82	99.18
	main		
1	0.10	95.82	4.07
2	0.00	65.68	34.32

7.4 Total Benchmark Execution Results

Although the previous results look very promising, Table 6 paints the picture of the total execution of the whole mcf benchmark. Furthermore, Table 7 shows the overall results of each helper threading scenario relative to the original benchmark. The instructions count shows the total cost of instructions, while the cycle count corresponds to the total execution time. Overall, the number of instructions is higher because of the implementation of helper thread. This is as expected. However, we hoped that these increased instructions would have resulted in a decrease in cycle count. Unfortunately, this is not the case.

Although the results around the delinquent load are very promising, the best overall improvement we achieved is with static sample based helper thread implementation, where decreased by 1.14% of the original execution time.

The helper process achieved almost the same result. However, it would be very difficult to improve the helper process any further. The result from the static sample based approach is far from what we expected. Because there is no sleep system call, we were forced to perform a tight loop inside the helper thread, until the main thread had reached a point from which the helper thread may once again begin prefetching. This tight loop results in a great increase in total instruction count. This increase in instruction count resulted, in turn, to an increase in total cycle count. Consequently, the static sample-base approach overall performed very poorly.

Table 6 mcf Statistic Over Total Execution

Helper Scenario	*# of Instructions*	*# of Cycles*
Original Benchmark	272854709	139307817
Loop-Based Process	285048826	142586354
Loop Base Thread	296634763	137718984
Sample-Base Thread	442788470	165584298

Table 7 Relative mcf Statistics Over Total Execution

Helper Scenario	*Inst. Increase (%)*	*Cycle Increase (%)*
Loop-Based Process	104.47	102.35
Loop Base Thread	108.72	98.86
Sample-Base Thread	162.28	118.86

8. DISCUSSION

There are essentially two purposes to the following section. First, we will summarize the results from Section 7. We will then provide some insight into why the results turned out as they did.

8.1 Summary of Results

Figure 12 provides a graphical representation of Table 7. We achieved improvements in L2 hits up to 71 % from the original benchmark using the sample based trigger. Because of the helper thread overhead and cache pollution gave negative impact of the improvements, we achieved, at best, a maximum of 1.15% overall cycle improvement.

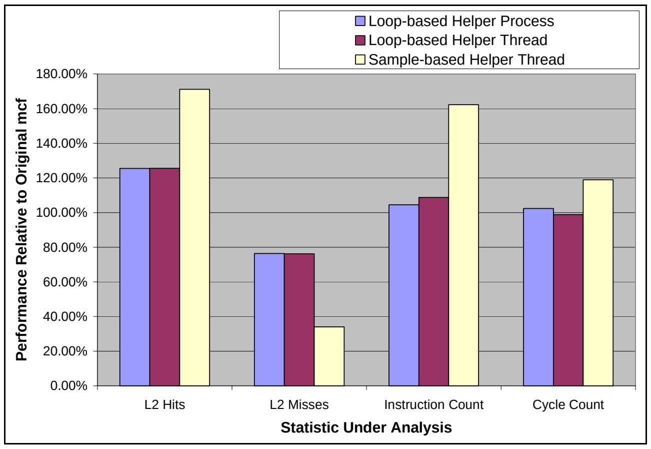


Figure 12 Relative mcf Statistics Over Total Execution

8.2 Discussion of Overall Performance

In our previous analysis of the cache access patterns of Phase 2, it is clear that static loop-based synchronization is limited by thrashing. That is, if we prefetch any more than 64 values into L2 cache before the main thread consumes them, they are swapped back out. This, of course, is detrimental to system performance. We were penalized for a cache L2 miss, the results of which are never used in main thread execution. Dynamic synchronization performance was limited by the fact that we had to have a tightly spinning loop to wait for the appropriate time to continue preexecution. A sleep system call implementation in the simulator would clearly improve these results.

Although we have not explored it in a great amount of detail, we speculate that phase 2 performance may be hindered by the mcf program control flow surrounding the delinquent loop. Referring to Figure 2, mcf phase 1 has 3716 iterations of the delinquent loop (a loop surrounding the delinquent loop, which is called from the primal_net_simplex function). Furthermore, mcf phase 2 has 2982 iterations of the delinquent loop. There are basically two control paths that can take us to another iteration of the delinquent load.

1. The first simply follows the normal incremental execution of the PC. That is, we enter the primal_bea_mpp function and eventually arrive at the delinquent load.
2. The second is a “goto” jump. Again, this jump is shown in Figure 2.

In Phase 1, 3487 of the 3716 delinquent loop iterations come from the first control path, and only 229 “goto” jumps are taken. In Phase 2, 1378 delinquent loop iterations come from the first control path, while 1604 come from “goto” jumps. Because the first control path simply follows the normal incremental execution of the PC, right when we enter the primal_bea_mpp function, it is possible predict the live-in variables and pass them to the helper thread. Hence, the helper thread is executed at the very beginning of the function entrance, quite far from the delinquent load. In the “goto” jump, the branch can only be predicted right after the previous delinquent load. Hence, we may not trigger the helper thread far in advance. We did not have a chance to explore this in any great detail, other than to acknowledge the behavior. We propose that this might be one of the complex synchronization problem which limited improvement in phase 2.

The project progress was also significantly impeded by the untrustworthy nature of the simulator, and the lack of suitable documentation.

9. FUTURE WORK

SSMT based on Simple Scalar v.3c does not implement a sleep system call yet. In our implementation, we had to do polling, which causes the number of instruction executed to significantly increase. This also increased the number of cycles, which effects the time of execution since the fetch bandwidth is also flooded with inter-context access. Overcoming this problem would probably involve bus synchronization and a cache coherence mechanism to be invoked in a very short interval.

Implementation of other helper thread scenarios, such as chaining, may provide very interesting and promising results. We speculate that chaining helper threads would likely improve performance when there is a small number of cycles between where the helper thread and the main thread must perform the same cache access.

Future work could also involve more detailed analysis on performance effects when miss latencies are assigned different values. As well, L2 cache size could be modified, and one could possibly try running the SMT processor with many more contexts.

Although we did not have time to implement it, future work could involve one master helper thread that monitors the memory access patterns of the main thread and assigns a child helper thread some region of memory as necessary. This would be a true dynamic approach, and could yield very exciting results. As well, the delinquent loop could contain several different delinquent memory accesses.

To improve performance results, a better understanding of the simulator is necessary to be able to debug the pipeline bottleneck. For example, one could determine the bottleneck in our static sample-based triggering method. Instead of improving the overall performance, this bottleneck in the pipeline resulted in the performance being much worse.

10. CONCLUSION

In conclusion, several scenarios for helper threading were implemented and analyzed on the SPEC2000 mcf benchmark. First, the delinquent load was determined. Then, three helper threading scenarios were implemented to prefetch the delinquent load into L2 cache. The first is a static loop-based helper process, followed by a static loop-based helper thread. Finally, a static sample-based helper thread was implemented. Performance results were obtained on the SSMT simulator. Our results indicate that sample-based triggering of helper threads results in the greatest improvement on the delinquent load, with a 71% increase in L2 hits and a 66% decrease in L2 misses, when compared to the single-threaded benchmark. However, sample-based triggering is very costly to the main thread as a whole. Consequently, loop-based triggering of the helper thread performs better on the benchmark as a whole, with only a mild speedup of 1.14%. Although these findings do not offer much in terms of overall performance benefits, they do lay down an excellent framework for future exploration in this topic. We hope that students in subsequent years continue this project to achieve some more impressive results. The source code for this project has been delivered along with the soft-copy of this document.

11. REFERENCES

[1] SPEC. SPEC CPU2000 Benchmarks. Standard Performance Evaluation Corporation, 2000. http://www.spec.org/osg/cpu2000

[2] D. Madon, E. Sanchez, and S. Monnier. A Study of a Simultaneous Multithreaded Processor Implementation. In *Euro-Par*, August 1999.

[3] D. Kim, S. S. Liao, P. H. Wang, J del Cuvillo, X. Tian, X, Zou, H. Wang, D. Yeung, M. Girkar, J. P. Shen. Physical Experimentation with Prefetching Helper Threads on Intels Hyper-Threaded Processors. In *Proceedings of the Second Annual IEEE/ACM International Symposium on Code Generation and Optimization with Special Emphasis on Feedback-Directed and Runtime Optimization (CGO2004).* San Jose, CA. March 2004.

[4] J. E. Smith, and G. S. Sohi, The Microarchitecture of Superscalar Processors, *Proceedings of the IEEE*, December 1995.

[5] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor , in *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996, pages 191-202.

[6] D. Burger and T.M. Austin, The SimpleScalar Tool Set, Version 2.0, *Technical Report 1342*, University of Wisconsin-Madison, CS Department, June 1997.

[7] Simulation and Modeling of a Simultaneous Multithreading Processor, D.M. Tullsen,In the *22nd Annual Computer Measurement Group Conference*, December, 1996

[8] Gautham K.Dorai, Donald Yeung and Seungryul Choi. Optimizing SMT Processors for High Single-Thread Performance. *Journal of Instruction Level Paralellism*, Vol.5, 1-35, April 2003.

[9] J. D. Collins , H. Wang , D. M. Tullsen , C. Hughes , Y. Lee , D. Lavery , J. P. Shen, Speculative Precomputation: Long-Range Prefetching of Delinquent Loads, Proceedings of the *28th annual international symposium on Computer architecture*, p.14-25, June 30-July 04, 2001, Göteborg, Sweden

[10] T. M. Aamodt, P. Chow, P. Hammarlund, H. Wang, and J. P. Shen, Hardware Support for Prescient Instruction Prefetch, *HPCA-10*, pp. 84-95, Madrid, Spain, February 14-18, 2004.

[11] A. Roth and G. S. Sohi. Speculative Data-Driven Multithreading. In *Proceedings of the 7th International Conference on High Performance Computer Architecture*, pages 191-202, Monterrey, Mexico, January 2001. IEEE.

[12] C. B. Zilles and G. Sohi. Execution-Based Prediction Using Speculative Slices. In *Proceedings of the 28thj Annual International Symposium on High-Performance Computer Architecture*, pages 2-13, Goteborg, Sweden, June 2001. ACM.

[13] G.K. Dorai, D.Yeung and S.Choi. Optimizing SMT Processors for High Single-Thread Performance, *Journal of Instruction-Level Parallelism*, Vol5, February 2003.

[14] B. Sprunt, *Pentium 4 performance-monitoring features*, IEEE Micro, Volume: 22 Issue: 4 , July-Aug. 2002 Page(s): 72-82.

[15] Intel Corporation. Vtune Performance Analyzer. http://developer.intel.com/software/products/VTune/index.html.

Appendix 1

This appendix is intended to be an informal detailed discussion of the difficulties encountered and improvements made upon the SSMT simulator.

The University of Maryland team had made some bug fixes and added many features over the original version of *SSMT* by [2] and used it for nice works like the one at [13] (to increase single-threaded performance). Apparently the one that is distributed by them as *SSMT* is the <tuned down> version of that work without all the extra algorithms they explored. Since this is not a major distribution like the *SimpleScalar* itself with many users, the possibility of more bugs exist. Also the fact that it comes only with *one paragraph* of documentation, made us to guess and explore how the proper way of launching multiple contexts and managing them might be. Often after following a clue, we ran to problems and figured out a way to hack it out.

Along the way, sometimes we learned how the intended usage could be and removed the unnecessary hacks. But there remains hacks for fixing bugs. Also, there are new code added for delinquent load profiling. The purpose of this appendix is to explain the problems we came along using *SSMT*, fixes we put and give an overview of code updates we made. This should make the life of anybody (if anybody!) following this project easier!

Very first trials

SSMT could be compiled smoothly. The only thing to be careful about is *not* running **make config-alpha** as the usual way with *SimpleScalar*. All the alpha based machine dependent files already exist in the main directory and SSMT changes are made over them only and not the ones in target-alpha. Running **make config-alpha** will replace the machine dependent files with *SSMT* support with the one without *SSMT* support from target-alpha directory.

We tried to figure out the version of *SimpleScalar 3.0*, *SSMT* is based on to be able to follow more easily the changes made on top of bare *SimpleScalar* for multi-threading support. Unfortunately *SSMT* updates are not marked or conditionally compiled. Comparing source code of *SSMT* against different versions of *SimpleScalar 3.0a*, *3.0b*, *3.0c* and *3.0d* didn't reveal much. We only figured out that it is definitely older than 3.0d and most likely based on 3.0c. Then we did some experiments using some prebuilt alpha binaries. Though no crash happened, the simulation result was all zero. Even the verbose mode did not show any instruction trace. After some debugging, it turned out that *SSMT* requires a special proprietary instruction (one of the special context instructions described below) embedded in the binary to trigger the simulation (*VSIM* instruction). Since we did not want to go through all trouble of changing the source code to add the required instruction (nor we had access to an alpha machine to build the benchmarks), we did a small hack in the simulator to get rid of this simulation mode activation (basically enabling the simulation mode right from the start of binary).

The first binaries we tried were *compress95*, *go*, *anagram* and *cc1*. To test the trust worthiness of *SSMT*, we compared simulation result of our binaries on *SSMT* (single threaded) against *SimpleScalar 3.0c*. It showed that though instruction count closely matched, there were large differnces for cycle count. For example for compress95 (SSMT against *Simplescalr 3.0c*), it was 226 against 86 for the first 20 instructions and 44132 against 48959 for the whole program. Digging to the simulator showed that it was because of different CPU configuration. SSMT (default settings) was using very aggressive hardware configuration (to support up to 32 contexts) so was using larger number of functional units and much higher fetch, issue and decode bandwidth. On the other hand, its cache latency was higher than *SimpleScalar*. As the result, SSMT was much slower for small number of instructions (because of larger delays in cache cold start). As the number of executed instruction was increasing, *SSMT*'s more aggressive hardware compensated its slower cache and finished the benchmark earlier. Below table compares the default hardware configuration of *SimpleScalar 3.0c* against *SSMT* for the settings that differ. All other settings were the same.

Configuration	SimpleScalar 3.0c	SSMT
instruction fetch queue size (in insts)	4	32
branch predictor type	Bimod	2lev
instruction decode B/W (insts/cycle)	4	8
instruction issue B/W (insts/cycle)	4	8
instruction commit B/W (insts/cycle)	4	8
register update unit (RUU) size	16	128
load/store queue (LSQ) size	8	64
L1 data cache	128:32:4:l	512:32:2:l
Unified L2 cache	1024:64:4:l	4096:64:4:l
Unified l2 cache hit latency	6	10
instruction TLB config	16:4096:4:l	1:4096:64:l
data TLB config	32:4096:4:l	1:4096:128:l
total number of integer ALU	4	8
total number of integer multiplier/dividers	1	2
total number of memory system ports available (to CPU) 2 / 8	2	8
total number of floating point ALU	4	8
total number of floating point multiplier/dividers	1	2

After changing *SSMT* settings to similar settings as *SimpleScalar 3.0c*, the cycle count for *compress95* reached very close to the cycle count of *SimpleScalar* (within couple of hundreds difference). This was good enough to show our *SSMT* setup and simulation environment was reliable. During this comparison process, we ported instruction logging (in verbose mode) as they cross the dispatch unit from *SimpleScalr 3.0c* to *SSMT* to make comparison through study of the logs easier.

Verifying delinquent loads with VTune

To verify that our profiling has found the right delinquent loads, we also managed to setup Spec2000 suite on Windows (after resolving some problems in building make tools and utility) and profiled cache misses of **mcf** with *Intel VTune Performance Analyzer*[15]. Not very surprisingly, we ended up with the exact delinquent loads as our own profiling with SSMT (as our cache latency and size configuration matches Pentium 4 configuration). Note that the result of *VTune* profiling is derived through sampling the program counter after a specified number of misses (e.g. every 1000 misses determined by *VTune* itself after its training phase) the estimation from sampling and is not the exact number of misses [14] but very close to the average behavior of application under profiling.

Investigating multiple context creation

Next we explored how to activate multiple contexts. Launching a thread through normal Unix system call was not an option as *SimpleScalar* didn't support fork and thread creation. When debugging the code, we noticed that there exists bit array of *context_config[]* accessed in *main()* controlling the total context number (*context_num*) and the context master array (*context_master[]*). Found the *SSMT* option of "**-context <array of active contexts>**" could activate multiple contexts but find it tedious as needed to pass an array of 32 parameters (one and zero). As we would need only 2 contexts for our target and we didn't have preference on which of the contexts (their indexes) would be running our main thread, we decided to extend **"-context_num <number>"** option to enable a number of contexts from context *0* to *<number> - 1*. Each one of these contexts requires its own process as when initialization finishes, instructions are fetched from program counter of all enabled contexts no matter if they have a proper process associated with or not (pretty weird!). For each context, a process can be specified to fetch from it at simulator start. Multiple processes name are passed to *SSMT* by the below format:

```
> ./sim-ssmt <proc1> <arg_1_1> ... <arg_n_1> ";" <proc2> <arg_1_2> ... <arg_n_2> ";" ...
```

A sample example for our usage was:

```
> <sim-ssmt -numContext 2 mcf.alpha  mcf_small.in “;” helperProcess  >& output_file
```

Note that use of a process requiring its inputs through input redirection (e.g. *compress95*) is a bit tricky here in this multi-process format as standard input is shared for all processes so only one of the processes can meaningfully use standard input through redirection.

The fact that we figured out a way to activate two contexts through launching two processes in the simulator directed us to use a form of helper process at first instead of helper thread. To be able to go forward, we needed to explore with special context instructions (explained below). So some sample test programs needed to debug multiple context activation. This was the point that being able to compile our own source code and generate alpha binaries become critical.

Problems with Spec2000 setup and compilation on the alpha system

After gaining access to the alpha system, we tried to setup *Spec2000* benchmark suite and compile the benchmarks. The prebuilt make utility and tool binaries coming with the suite didn't run properly on our system (*Tru64 Unix*). We had to resort to building them from the provided tools source code in the benchmark. For couple of the tools we received compilation errors because of some conflicts with existing Compaq compiler's include files. Had to make minor changes in the tools source code to make it compatible and setup the make/utility binaries. All benchmarks we built crashed on both simulators (*SSMT* and *SimpleScalar*) with "Segmentation fault" while they were fine on the alpha machine itself. Debugging on the simulator turned out that it was because of the system calls issued by the binaries. The default master makefile of the benchmarks use dynamic linkage with C and system libraries but simulator requires static linkage as the libraries doesn't exist on the host machine. Also, the makefile updated to produce debug version of benchmark to facilitate matching machine code against the source code (when matching the position of delinquent loads extracted through profiling against the source code).

SSMT's context instructions

SSMT introduced a series of context related instructions to extend alpha ISA. To be able to compile with a normal compiler, it exploited PERR (Pixel Error) instruction with different combinations of register parameters to decode different proprietary context instructions. This instruction is unused for most architecture/applications and is treated as a NOP on our machine. For example, LCU is defined through **"perr %0, %1, $13"**, so any PERR instruction which uses register 13 as its third parameter is treated as an LCU in the simulator and its proper handler will be called (associated through *machine.def* macros like other instructions). The first and second registers denoted by **%1** and **%2** can be any register and their content will be used as parameters for LCU. See *ssmt.h* file for the list of all these instructions. This file has wrapper macros that was supposed to be included in the source code (e.g. our benchmark) to facilitate use of context instructions. We had to modify it as it was using *gcc*'s inline assembler format which didn't conform to our Compaq compiler and was causing compilation error.

From all different context instructions provided, we were mainly interested on context suspension and resumption instructions. Since there wasn't a manual, we had to investigate the source code and guess what instructions might do such tasks and what could be the proper usage sequence of them. It turned out that LCU, LPCR and SPCX are the most interesting instructions for our case. LCU has a format like **"LCU(port, context)"** where port parametrises its function, so it can be used for both suspension and resumption (from the previous suspension point) of a particular context through any context. SPCS has a format like **PC = SPCS(context)** where it suspends a context and returns the PC it is stopped at. This PC can be passed to LPCR like LPCR(context, PC) to resume it again. We will see later that LPCR is part of the mechanism to start a new thread without using a previous PC.

Context management in SSMT and hacks we made

SSMT has defined a form of context management register (*regs_ACXT*) with a global nature (not part of each context's register file). Context instructions change a context's enable bit in this register when they're decoded in *ruu_dispatch*. When *regs_ACXT* changes, the proper context bits in *sim_restart_context* and *sim_shutdown_context* variables are enabled for every new context resumed or suspended respectively. These variables behave like a one-shot trigger where they're cleared back to 0 in the issue stage (*ruu_issue*) and instead *sim_active_context* is updated to keep track of active contexts and help other stages (like fetch) to adapt their function (like degree of partitioning). Basically, the context instructions are committed in the issue stage where the program counter for resumed contexts are updated.

We found several issues with using LCU and LPCR instructions. When LCU suspends a context, it backs up the program counter of that context. This program counter is supposed to be the PC of the next instruction to be executed by that context. In the special case that a context suspends “itself”, the next PC variable is not updated yet resulting in saving the current PC where the suspending LCU instruction itself resides. Now when another context wakes up the suspending context, the context is reactivated at the PC of the old LCU instruction again causing no resumption. Another issue was with handling of context resumption in *ruu_dispatch* where *restart_context* variable was not set properly. For more

information, see the new code added towards the end of *ruu_dispatch()*. The other issue with LPCR is explained below in

<Create Thread> section.

New system call implementation

For the helper process method, main process needs to share the live-in variables with the helper process through some form of inter-process call. Since such a standard system call is not supported by *SimpleScalar*, this is done through two new system calls. The implementation is easy. The system call handler in *syscall.c* is extended to handle two new system calls *OSF_SYS_SetPrivateData* and *OSF_SYS_GetPrivateData*. The data to be shared is loaded into a0 to a4 registers and then *Sys_SetPrivateData* call made by the main process. These data are stored in some simulator global variables. When the helper process calls *Sys_GetPrivateData*, the data is retrieved through filling the registers back. Note that system calls are not expensive on *SimpleScalar* as they accounted as one instruction only and only their final effect (e.g. printing to host machine at *printf*) is simulated on the host machine.

Memory hierarchy and the hack for MMU

The first time we tried the helper process method (explained in section <X>), the helper process didn't help at all. Basically the main thread was incuring L2 misses on most delinquent loads though that memory was prefetched by the helper process a short time ago. First guess was that somehow cache trashing is evicting data loaded by helper process before consumption by the main thread (the prime suspect was the code within the body of the loop in main thread after the delinquent load). After investigating the log of all memory accesses, that possibility ruled out. After debugging the simulator, it turned out that cache lines are also tagged with the context ID of requesting context. This was done since *SimpleScalar* uses virtual address cache tagging rather than physical address tagging. So everything loaded by the helper process into the cache was not used by the main thread at all as they had different context tags. The problem was fixed by hacking all memory accesses initiated at the prefetch area of helper thread to present them as if they were issued by main thread's context.

Speculative execution problem

Speculative execution feature of *SimpleScalar* gave us a hard time for a while. We had added some trace messages to SSMT in handler functions of context instructions to monitor the sequence they were issued by our modified benchmark. We noticed that after most of benchmark calls to LCU context instruction in the main thread (to trigger the helper context), the simulator prints out the trace message of another context instruction which was never called in the whole benchmark! After lots of debugging, it turned out that the simulator's speculative execution could go beyond a function's return instruction (pretty weird!) to the code of next function residing below it!! We had a bunch of wrapper functions for context instructions defined after each other (see *helper.h*). The benchmark was calling LCU wrapper but simulator speculated beyond that function code executing instruction from the other wrapper residing below it and printing our the trace message as the result. Since this was only speculative execution, it wasn't hurting the simulation accuracy but was making the tracing very confusing. We decided to disable speculation by turning off branch prediction using a perfect predictor. So all the reported results are based on a perfect predictor.

Thread creation

As explained in <section X>, we decided to use the thread creation mechanism provided in *SSMT*. Since such a system call was not implemented in *SimpleScalar*, SSMT provided some helper functions to be used in applications instead of supporting thread creation system call and translating it to its own context mechanism. Below code shows the helper function *createThread* to be called by the application's main thread and *_startThread* which is the new thread's entry point before calling the thread function.

```
void createThread(__int64 threadId, __int64 context, __int64 stack,  __int64 func, __int64 params)
{
   #define port 0x1000

  ((unsigned long long *)stack)[-7] = (unsigned long long)threadId;
  ((unsigned long long *)stack)[-6] = (unsigned long long)params;
  ((unsigned long long *)stack)[-5] = (unsigned long long)func;
  asm (".set    noreorder");
  asm ("stq $29, -32(%0)", stack);         // $29 is gp
  asm (".set    reorder");
  SSMT_LPCR(context, ((__int64)_startThread)  + 12);
  SSMT_PMSG(port, stack);
}
```

```
void _startThread()
{
    asm (".set    noreorder");
    asm("mov     0x1000,$sp") ;            //     # li      $sp,0x1000
    asm("perr    $sp,$sp,$18") ;  //     # gmsg    $sp,$sp
    asm("ldq     $16,-56($sp)") ; //     # threadID
    asm("ldq     $17,-48($sp)") ; //     # argv
    asm("ldq     $27,-40($sp)") ; //     # function to execute
    asm("ldq     $29,-32($sp)") ; //     # gp register
    asm("subq    $sp,96,$sp") ;            //
    asm("jsr     $26,($27)") ;             //     keeps return address in r26 and jumps to r27 which is func
    asm("ldq     $0,40($sp)") ;            //
    asm("addq    $0,0x1000,$0") ; //       # termination port
    asm("perr    $31,$31,$13") ;  //     # lcu     $0,$0 # endless wait
    asm (".set    reorder");
}
```

The followings are passed to *createThread*:

- <*threadId*> can be any value picked by the caller for purpose of identification only.

- <*context*> is the context index this thread will be started on,.

- <*stack*> is the memory region to serve as the stack of the new thread. Caller (main thread) needs to allocate a proper memory size (large enough depending on the complexity of the thread function) and pass the address of bottom of the allocated memory as the stack grows towards the lower addresses.

- <*func*> is the address of new thread's entry function.

- <*params*> is the pointer to a parameter area to be passed to the thread function.

createThread creates a proper stack frame in the stack memory, then activates the new thread context through LPCR instruction. The thread doesn't start right from the thread function and goes through *_startThread* first. This is to pass the stack pointer to the new thread and also make sure the thread is started before main thread continues. *_startThread* first makes a GMSG call (one of the special context instructions) to receive the message posted by the main thread. The main thread posts this message by PMSG instruction right after its LPCR call. The message contains the stack pointer for the new thread which there is no other way to pass it to the new thread (note that the new thread starts within its new context and doesn't inherit the content of main thread's register file). PMSG and GMSG are blocking instructions, so the main thread blocks till new thread executes the GMSG instruction, so it is used for an implicit synchronization too beside passing stack pointer. The <port> parameter is just a message identifier and could be any value mutually used by *createThread* and *_startThread*. After setting its stack pointer, the new thread loads its argument registers from proper stack locations and jumps to the thread function. When thread function ends, it just suspends itself by making a LCU call. Note that since only one process is used in this helper thread scenario a normal launch of simulator as below is enough without the need for *context_num* and the helper process:

```
<sim-ssmt mcf.alpha  mcf_small.in  >& output_file
```

After implementing such a sequence in a sample test program, we figured out that the new context is not really activated as simulator just activates the context bit and doesn't know that number of active contexts has changed. Even fixing this, was not enough as the new context didn't have any cache, TLB, MMU (a *SimpleScalar* object managing memory hierarchy and paging), branch prediction,... associated with. As a simple fix, we shared all of them with the caller's context i.e. the main thread (knowing that in a real system some other resources are partitioned or duplicated) This hacks solved the problem at the end. At a later point we realized that a more proper way of doing all this was to start the simulator with two contexts right from the beginning but pass a special dummy process for the second process. This process is activated in the second context in the beginning but suspends itself right away by executing a single LCU instruction. When we launch a new thread, it will use the already initialized context but will jump to the thread function. Even in this scenario we still need to hack MMU to prevent tagging of cache lines with the context ID as explained before.

Source code changes log:

Below briefly explains the changes made to SSMT source code. All are tagged with an *#ifdef ECE1718*, so can be easily identified from the original code.

Makefile: Defining *ECE1718* compile flag.

cache.c: Setting global profiling variables for tracking the last evicted cache line and its miss type.

machine.c: Adding definition of *md_xor_regs()* function from SimpleScalar3.0c for printing XOR of register file content (helpful for debugging).

machine.h: Decleration of *md_xor_regs()*.

main.c: The hack for **"-context_num <n>"** command line option to activate multiple contexts indexed from 0 to n-1 rather than relying on more generic **"-context"** option which requires a array of 32 context IDs.

syscall.c: Implementation of new system calls *OSF_SYS_SetPrivateData, OSF_SYS_GetPrivateData.*

sim-ssmt.c:

Definition of PC of delinquent load (in the main thread) and prefetch load (in the helper) to be profiled.

Definition of all profiling arrays for recording all data accesses made at delinquent/preftch loads at both main and helper sides.

- Changing cache configuration and making it like Pentium 4 settings in *sim_reg_options()*.

- Printing instruction commit progress (every 10000000) in *ruu_commit()* to detect if simulator is dead

(necessary for our long running simulations).

- Adding cache profiling and hacking ASID (Address Space ID) in *ruu_issue()*.

- Printing invalid instruction in *ruu_dispatch()*. Note that if you are not using perfect branch prediction, you might want to turn this off as *SimpleScalar*'s speculative execution sometimes reach invalid instructions and these extra messages become annoying.

- Dumping out instructions passing through dispatch unit (in *ruu_dispatch()*) if verbose mode enabled.

- Fixing the bug for restarting a suspended context (setting *sim_restart_context*) in *ruu_dispatch()*.

- Getting rid of the dependence on the special VSIM instruction for activating simulation (in *sim_main()*).

- Printing cache profiling results (main and helper thread miss counts; all data addresses accessed by

delinquent/prefetch loads in main and helper threads).

- Moving all handlers for context related instructions to *sim-ssmt.c* from *machine.def*. This makes debugging

much easier (e.g. setting breakpoints) as *machine.def* was included as a header file within *sim-ssmt.c* and

couldn't put breakpoint in that file.

- Tracing code to track issue of context instructions.

- Fixing bad update of program counter for LCU instruction handler in *load_cxtunit_reg()*.

- Hacking LPCR to share TLB, cache, return stack, branch predictor, MMU and context address range

(e.g. *ld_text_base, ld_text_size, ld_data_base, ld_stack_base*).

helper.h: This is a limited replacement for *ssmt.h* and *ssmt.c* we created to make our life easier using context instructions. It is meant to be used by the application (e.g. benchmark) as a replacement for *ssmt.h* and *ssmt.c* as the Compaq assembler didn't accept *gcc*'s inline assembler directives. functions