ParaWeaver: Performance Evaluation on Programming Models for Fine Grained Threads

Jiayuan Meng, Dee A. B. Weikle, Kim Hazelwood
University of Virginia

Abstract

There is a trend towards multicore or manycore processors in computer architecture design. In addition, several parallel programming models have been introduced. Some extract concurrent threads implicitly whenever possible, resulting in fine grained threads. Others construct threads by explicit user specifications in the program, resulting in coarse grained threads. How these two mechanisms impact performance remains an open question. Implicitly constructed fine grained threads exhibit more overhead due to additional thread scheduling, thread communication, and thread context switches. However, they also increase the flexibility in scheduling. Therefore, computation resources can be utilized further and workloads are more balanced among cores. Moreover, if scheduled properly, concurrent fine grained threads may exhibit more data affinity than coarse grained threads. In most parallel architectures, the last-level cache is typically shared among all the cores. Therefore, it is exposed to contention and pollution due to concurrent threads. As a result, data sharing becomes important. A greater degree of data sharing among threads results in fewer last-level cache misses, which is one of the main latencies for a multithreaded process. The data-sharing behavior among the threads depends on how the applications are parallelized and how the threads are scheduled. The complex nature of many applications leads to nested structures in the call graph, and concurrency can be found from a course grained level to a fine grained level. In this project, we compare the data sharing behavior of coarse grained threads and fine grained threads, and evaluate their performance on a CMP cache simulator.

1. Introduction

We extract fine grained threads by regarding each function call and each loop iteration as a potential thread. Dependency is analyzed among the threads. Coarse grained threads are defined manually by programmers using code annotation based on Pthread functions. Performance is compared between the two methods.

Over the previous decades, various work has been done on extracting concurrency from common applications. Languages like Multilisp [11] and X10 [3] have been proposed. Application programming interfaces for parallel programs such as Pthreads, OpenMP, and MPI are becoming more and more popular. Some of these programming models are able to expose concurrency to a fine-grained level, while others are meant for concurrency at a course grained level. As hardware evolves to the era of chip multi-processor(CMP), the communication cost becomes lower and more data sharing among the threads are beneficial. Jaleel et al. [6] found that on a typical CMP design, a shared last-level cache(LLC) outperforms private last level caches. Their conclusion is drawn from experiments on MineBench, a parallelized benchmark suite for data mining applications. We take the next step and evaluate the influence of thread extraction and thread scheduling on data sharing behavior among the threads. We base our evaluation on the SPLASH2 benchmarks, programmed with Pthreads. We regard threads defined in these benchmarks to be coarse-grained threads. For fine-grained threads, we envision that in the near future, it may be possible to have a programming model that treats every procedure and every loop iteration as a potential thread. For a parent thread, dependencies among its child threads are known statically or can be speculated. This programming model supports fine grained threads, and we are interested to compare it with the same benchmarks implemented as coarse grained threads using Pthreads. Because no available applications are coded with fine-grained threads described above, we perform a trace analysis and extract the fine-grained threads from the trace of a sequential program. We call threads extracted in this way **Pseudo Trace-based Threads**, or, **PTthreads**. To make a valid comparison with coarse grained threads, we modify the SPLASH2 benchmarks to be sequential programs, which use annotations rather than Pthread library calls to indicate where to fork or join a coarse grained thread. In this way, coarse grained threads are extracted as PTthreads as well, and are used to analyze data sharing behavior on a combined CMP scheduler and cache simulator. We call out tool **ParaWeaver**. We evaluate the benchmark of fast fourier transform and find fine grained threads exhibit more data sharing among concurrent threads.

Section 2 presents some recent work in workload characterizations. Section 3 describes our assumption about a programming model for fine grained threads. Section 4 describes the ParaWeaver tool that we use to extract PTthreads from a sequential program. section 5 demonstrates our results. We conclude our findings in section 6.

2. Related Work

Eeckhout, Sampson, and Calder [5] use the micro-architecturally-independent characteristics outlined in Phansalkal[10] to analyze the SPEC2000 benchmark suite written in legacy code. While micro-architecturally-independent characteristics reveals significant properties of sequential programs, little work has been done on characterizing parallel programs. Jaleel et al[6] have characterized MineBench benchmarks based on their cache sharing at the last level cache(LLC). The characterization reveals the importance of a shared LLC. We base our approach on this shared LLC assumption.

To study the properties of parallel programs, Asanovic et al. have identified 13 dwarves they think cover the main categories of parallel applications [7]. These benchmarks have various parallelism and communication patterns. They are of specific interest to the parallel architecture designers. We pick several important benchmarks from their selection.

Rul et al.[12] have studied function-level parallelism for bzip2. Their method is similar to ours in that they also analyze traces captured from a running process. In their methodology, a call graph is captured. Besides the call graph, a special data dependency graph is constructed to provide information about concurrency. The data dependency graph has nodes for both functions and data. Data-nodes are connected to all functions that share it. In this way, data sharing is easily detected from the dependency graph. The functions are then regrouped into threads according to how much data they share with each other. They actually construct real threads and run them on a multiprocessor with 4 cores. Their result shows a maximum speedup of 3.64. While they achieved good performance for bzip2, their methodology only focus on function level parallelism of bzip2, while we want to explore fine grained threads on a more complete set of typical parallel programs. As in Rul et al.[12]'s work, our analysis focuses on memory dependencies, since register dependencies can be predicted[13] or precomputed[4]. However, we are not attempting to translate a sequential program to a real parallel program. Instead, we focus on characterizing the effect of fine-grained threads on cache sharing behavior and thread communication.

In addition, several programming models have recently been introduced that expose parallelism at fine granularity, including data parallel haskell[2], shader metaprogramming[9], and NVidia's Cuda programming toolkit. Nevertheless, fine grained threading tends to require more effort from both the software side and the hardware side. Moreover, it usually demands more effort from programmers. Our work attempts to justify this effort in fine grained threading.

3. A Programming Model for Implicit Fine Grained Threads

We propose a programming model that extracts fine grained threads implicitly. The assumption is that the programming model regards each function call and each loop iteration as a potential thread. With appropriate programmer hints, compiler analysis and hardware speculations, the dependency among the potential threads can be pre-computed or predicted. We base our definition of fine grained threads on function level parallelism and loop iterations for the following reasons:

1. Often, loops carry no dependencies among iterations and can be expressed as "foreach". Pthread and OpenMP are commonly used to extract parallelism based on loop iterations.

2. Functions exhibit a nested function-call structure. Leaf funcions can be easily defined as light-weight kernels that fit fine grained threading.

3. Functions are natural units for computation grouping. Closely related instructions with a significant amount of dependencies tend to be grouped within a function.

4. Functions are conceptually the smallest unit of user configurable tasks. A programmer tends to configure or optimize computation based on functions. We expect the parallel programming model to allow programmer to express additional information based on functions. This additional information may include data access patterns(which region of the array is going to be consumed), data sharing status(read-only or read-write), etc. With this information, more dependencies can be deduced statically, and run-time thread speculation becomes more accurate.

5. Previous work[1][12] has shown that function level parallelism demonstrate good speedup. Balakrishnan and Sohi et al. [1] made an effort to speculatively execute methods. In their approach, programmers pick which method to execute speculatively. The appointed method usually performs a significant amount of computation. Their result also shows good speedup, which supports function level parallelism.

4. Methodology

Since our dependency analysis is based on memory dependency, dependencies caused by side effects are omitted. As a result, the user must annotate the source code to indicate the interested code region for our analysis. The selected code region is typically computation and data intensive. It forms the main component of the program's computation with no side effects (such as printing). We are not counting register dependencies since many register dependencies are caused by hardware limitations. Most importantly, register dependencies can be predicted[13] or precomputed[4].

The annotated code is compiled by gcc/g++. We use Pin [8] to capture the memory trace of a process. Because dynamically linked libraries may be linked to a program with a far jump and no returns, we require programs to be statically linked so that Pin can capture all procedure calls and returns. For each instruction, we record in the trace its program counter, load address and store address. Information about loops is recorded as well. In addition, user markers are preserved in the trace. After that, the trace record is post-processed to extract PTthreads based on procedure calls, loops, and user markers. PTthreads are organized in a nested task graph. Dependencies among child PTthreads are known to the parent PTthread. With appropriate memory reallocation, PTthreads can be rescheduled and simulated on a CMP cache simulator. Statistics can then be gathered and analyzed.

Figure 1 gives a birds-eye view of what ParaWeaver does.

4.1 Source Level Interaction

Besides annotating the interested code region to inspect, we also rely on programmers to define coarse grained threads. In reality, it is common for programmers to define threads explicitly using Pthreads, OpenMP, etc. In our experiment, we replace Pthread function calls in SPLASH2 benchmarks with user markers. By doing this, we can make sure that our definition of coarse grained threads corresponds to reality.

We provide five markers for code annotation. These markers are merely names to functions with no-ops. Pin is able to capture procedure calls with procedure names. Thus, user markers can be captured and analyzed. $THREAD\_BEGIN$ and $THREAD\_END$ enable programmers to regroup the computation that can potentially form a thread. Programmers can also insert a $THREAD\_BARRIER$ marker whenever they want all the previous child threads to finish before continuing the parent thread. This marker is needed when there is a dependency among the threads so that the consumers have to begin execution after the producers finish.

To annotate the beginning and end of the interested code region, programmers can use $PROCESS\_BEGIN$ and $PROCESS\_END$.

Figure2 shows simplified sample code for a fast fourier transform with code annotation.

4.2 Loop Detection

Loops are detected based on backward branches and other heuristics. Instruction traces between the backward branch target address and the most recent instruction at its branch target address forms an iteration. To extract PTthreads from an iteration, we need to isolate the main loop body from the loop monitor code which always causes a dependency across iterations. To do this, we maintain a small window when tracing the program under Pin. Each dynamically executed instruction is given a unique ID which increments per instruction. A window records memory load information for the most recent instructions and their dependent stores. When a taken backward branch is found, we compare the current instruction's ID to the store instruction's ID recorded in the window. If the store instruction is within a reasonable distance (4 instructions, in our example), we will isolate the code between the store instruction and the current branch instruction as loop monitoring code. We realize that this loop-body isolation technique doesn't cover all cases and is not precise, but experiments show it works for the most common scenarios.

Figure 3 shows an example of loop detection. The detected backward branch corresponds to line 17 in the FFT source code. In this example, the loop monitoring code in bold font is detected and isolated from the loop body.

4.3 Synthesize Memory Trace for a Parallel Program

As an offline post process, ParaWeaver loads in the sequential memory trace and synthesizes memory traces of a parallel program. A call graph representation of the PTthreads is built as well. Each node in this call graph represents a PTthread. Dependencies among the child PTthreads are kept within each node.

Figure 4 illustrates a PTthread node in the call graph. Shaded areas represent child PTthreads that link to child nodes. White areas represent native computation within this PTthread. A dependency graph among native computation segments and child PT-

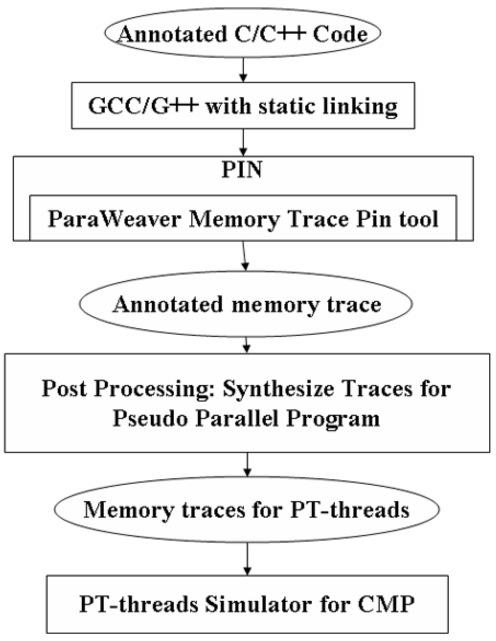
Annotated C/C++ Code
GCC/G++ with static linking
PIN
ParaWeaver Memory Trace Pin tool
Annotated memory trace
Post Processing: Synthesize Traces for Pseudo Parallel Program
Memory traces for PT-threads
PT-threads Simulator for CMP

Figure 1: Flow Chart of ParaWeaver

threads also resides in each node (not shown in Figure 4). Note that the per-node dependency graph is built by traversing the sequential memory trace while simulating a call stack. Memory loads and stores issued by the current threads are also broadcasted to parent threads in the call stack. Dependency edges are added accordingly. Figure 5 gives a simple example of the call graph.

To compare the fine grained threads and coarse grained threads, we provide two options for building the call graph and extracting PTthreads. PTthreads can be extracted where programmers annotate the source code. This correspond to the extraction of coarse grained threads. Fine grained threads are extracted from procedure calls and detected loops.

4.4 Detecting Shared Variables and Private Variables

The PTthreads extracted directly from a sequential memory trace cannot be used to study the cache sharing behavior of a parallel program. In a sequential program, variables are often reused by multiple statements, even though these statements may actually be turned into parallel statements. A typical example is a loop iteration with iteration-private variables. Some common scenarios are:

```
float a;
for(int i=0; i<MAX; i++){
    a = foo(i);
    ...
}

for(int i=0; i<MAX; i++){
    float* a = (float*)malloc(sizeof(float)*100);
    foo(a);
    ...
    free(a);
}
```

Assuming in these scenarios, iterations are independent with each other (no RAW dependencies across iterations). A PTthread will be extracted for each iteration. These PTthreads may execute in parallel when rescheduled on our CMP cache simulator. Although variable *a* is reused by each iteration in the legacy code, it will be incorrect to reuse *a* in the parallelized code. Instead, *a* should be a private variable to each iteration if iterations are treated as concurrent treads. Therefore, we have to distinguish between variables that are shared by concurrent PTthreads, and variables that are reused and are actually private to concurrent PTthreads. For reused but private variables, we will have to rename the variable to another memory address.

The algorithm contains two passes. In the first pass, PTthreads are extracted along with their dependencies. Every store records which PTthread is the previous producer for the same address. In the second pass, we go through the trace in sequential order again. For each store, we compare the PTthread it belongs to and the PTthread that writes to the same address previously. If the two PTthreads have dependency, then the data at this address is shared. Otherwise, the data is PTthread private and has to be renamed. Every load instruction updates their load address to the most recently renamed address.

We need two passes because when a PTthread writes to an address that has been modified by a previous PTthread, it doesn't know whether the other PTthread has dependency with itself until the current PTthread is fully built. Since dependencies are recorded locally (a parent thread only knows the dependency among its child threads), it is not straightforward to tell whether two arbitrary threads have dependency. A naive approach is to construct a full graph among all the threads, which is both time and space consuming.

We take another approach and use the call stack. Since PTthreads are traversed in a depth first order in the sequential program, each PTthread is assigned to an ID according to this order, as illustrated in Figure5. Suppose PTthread 8 writes to an address which is previously modified by PTthread 4. At the same time, the current call stack contains PTthread 6 and PTthread 1. Since only the parent node knows the dependency among the child nodes, the visible dependencies are those between PTthread 2 and 6, and PTthread 7 and 8. To decide whether PTthread 8 and 4 are dependent, we need to find which PTthread in the call stack is the most recent ancestor shared by PTthread 8 and 4. This is easily computed by finding the lower bound of ID 4 in the call stack. In this case, it is PTthread 1. We also need to find which child of PTthread 1 has PTthread 4 as its descendant. This is achieved by finding the lower bound of ID 4 in the list of the child PTthreads of PTthread 1. In this case, the child list of PTthread 1 contains PTthread 2 and 6. The lower bound for PTthread 4 is 2. To decide whether PTthread 8 is dependent on PTthread 4, we only have to look up whether PTthread 6

```
1:   PROCESS_BEGIN();
2:  while (n > mmax) {
3:        wtemp=sin(0.5*theta);
4:        wpr = -2.0*wtemp*wtemp;
5:        wpi=sin(theta);
6:  for (m=1;m<mmax;m+=2) {
7:              for (i=m;i<=n;i+=istep) {
8:                   THREAD_BEGIN();
9:                      j=i+mmax;
10:                    tempr=wr*data[j]-wi*data[j+1];
11:                    tempi=wr*data[j+1]+wi*data[j];
12:                     data[j]=data[i]-tempr;
13:                     data[j+1]=data[i+1]-tempi;
14:                     data[i] += tempr;
15:                    data[i+1] += tempi;
16:                  THREAD_END();
17:              }
18:              wr=(wtemp=wr)*wpr-wi*wpi+wr;
19:              wi=wi*wpr+wtemp*wpi+wi;
20:        }
21:        mmax=istep;
22:     THREAD_BARRIER();
23:  }
24:  PROCESS_END();
```

Figure 2: Annotated source code for fast fourier transform. Functions in bold font are annotations

```
i 804930c    r    8b4df48:4
i 804930e    r    bfd954c4:4
i 8049311    w    8b4df48:4
i 8049313    w    bfd9545c:4
i 8049318    r    bfd95488:4
i 804931b    r    bfd9548c:4    w    bfd9548c:4
i 804931e    r    bfd9548c:4
i 8049321    r    bfd95478:4
i 8049324
i 8049249    w    bfd9545c:4
i 804924e    r    bfd9547c:4
i 8049251    r    bfd9548c:4
```

Figure 3: Loop Detection For FFT source code around line 17.

is dependent on PTthread 2, which is recorded in PTthread 1. This algorithm requires no extra storage and has the time complexity of O(log(NM)), where *N* is the depth of the call stack and *M* is the number of child PTthreads of the shared ancestor.

4.5 Synthesizing a Parallel Process

After ParaWeaver constructs the call graph and synthesized memory trace for PTthreads, it reschedules the PTthreads and simulates an execution of a parallel program. Users are able to specify the number of cores, the scheduling policy, and memory access policy. The available scheduling policies are FIFO and LIFO. FIFO scheduling corresponds to breadth first traversal of the call graph, while LIFO scheduling corresponds to depth first traversal. For memory accesses, ParaWeaver provides three policies. A core can either stall and wait for a memory request, or it can continue execution assuming a perfect MSHR implementation. Another option is to switch to another PTthread whenever a memory request is issued, which is the scheduling policy implemented in some graphics hardware.

We choose our simulation hardware to have a two level cache hierarchy. Each core has its private 16K L1 cache, the L2 cache is shared and has a size of 512K. Memory requests to the L2 cache are uniform. Memory is also modeled. Figure 6 illustrates the structure of our CMP cache simulator.

Our simulation is not cycle accurate. It assumes a uniform cycle per instruction(CPI) for instructions don't access memory. Overhead caused by thread context switches are tunable in our simulator. Although the simulator is not cycle accurate, we can still gain some insights from the experiment. In our simulation, consumers will always start when its producers finish execution.

5. Results

Figure 7 shows the speedup normalized to coarse gained threads running on one core with FIFO scheduling. Threads also stall on memory requests. Figure 8 shows the same result but plotted with lines. For last level cache miss rate, Figure 9 and Figure 10 show the results for the last level cache miss rate.

Our results indicate that fine grained threads have significant speedup than coarse grained threads on CMP architectures. Even for this simple FFT program, the speedup usually ranges from 1.5 to 2 regardless of the scheduling policy and memory request mechanism. Moreover, fine grained threads have a lower shared cache miss rate, as we expected. FFT is a micro-benchmark that a fine-grained threaded program doesn't differ significantly from a coarse grained program. In the case of FFT, L2 cache miss rate under fine grained threads doesn't differ much from coarse grained threads. Therefore, we believe that the performance gains come primarily from the fact that workloads are more balanced among the cores under fine grained threads than from coarse grained threads. We think that the benefits of fine grained threads will become more obvious for more complex programs with a deeply nested call graph

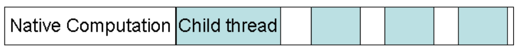

Figure 4: A Typical PTthread node in the call graph

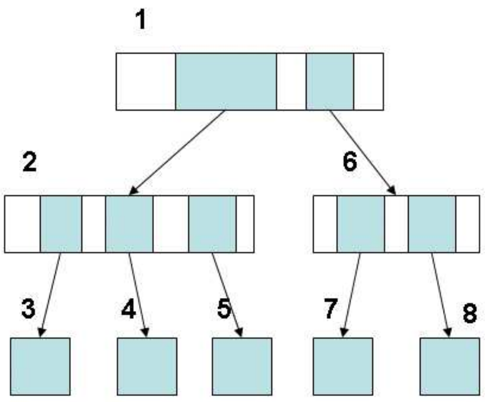

Figure 5: Call Graph of PTthreads

like video processing, where threads can be extracted from each frame, and from each pixel in one frame.

Our results also show that FIFO scheduling and LIFO scheduling don't make much difference for the FFT application. Except for coarse grained threads, if the core switch to other threads whenever a memory request is being made, FIFO scheduling usually performs better than LIFO scheduling in both speedup and L2 cache miss rate. This result is counter-intuitive, because LIFO scheduling tends to execute concurrent threads with similar working sets. The fact that LIFO doesn't make much difference may be due to the fact that FFT has a flat call graph with doesn't exhibit a deeply nested structure, all the threads are mainly extracted at the same layer. Thus, LIFO and FIFO don't make much difference. FIFO scheduling has advantages over LIFO scheduling in that FIFO scheduling corresponds to a breadth first traversal of the call graph and can spawn more threads than LIFO scheduling, which corresponds to a depth first traversal of the call graph.

6. Conclusions and Future Work

In this paper, we show that a programming model for fine grained threads is beneficial to parallel programs running on CMP architectures. This is even true for a our microbenchmark which has a flat call graph. We expect that for a more complex program with a deeply nested call graph, fine grained thread's benefits will become more obvious for the following reasons:

1. Fine grained threads expose parallelism into much finer details than coarse grained threads. Therefore, it is easier to balance the workload among several cores.

2. If scheduled appropriately, fine grained threads can be executed in a way that concurrent threads have better data affinity and share a significant section of working set. This will lower last level cache misses.

However, we haven't evaluated the impact of increased thread communication and increased thread context switches due to fine grained threads. As our future work, we will characterize the impact of these factors. More sophisticated benchmarks will be analyzed as well.

In summary this work indicates fine-grained threads will be beneficial if communication of switching overheads can be kept low enough.

7. Acknowledgements

This work is supported in part by the National Science Foundation under grant nos. NSF CAREER award CCR-0133634, and CNS-0340813, and a grant from Intel MRL. Great appreciation goes to Dan Upton for his helpful input.

8. References

[1] S. Balakrishnan and G. S. Sohi. Program demultiplexing: Data-flow based speculative parallelization of methods in sequential programs. *SIGARCH Comput. Archit. News*, 34(2):302–313, 2006.

[2] M. M. T. Chakravarty, R. Leshchinskiy, S. P. Jones, G. Keller, and S. Marlow. Data parallel haskell: a status report. 2007.

[3] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM Press.

[4] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen. Speculative precomputation: long-range prefetching of delinquent loads. In *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*, pages 14–25, New York, NY, USA, 2001. ACM Press.

[5] L. Eeckhout, J. Sampson, and B. Calder. Exploiting program microarchitecture independent characteristics and phase behavior for reduced benchmark suite simulation. In *IISWC05*, Oct. 2005.

[6] A. Jaleel, M. Mattina, and B. Jacob. Last-level cache (llc) performance of data-mining workloads on a cmp–a case study of parallel bioinformatics workloads. In *Proceeding of 12th International Symposium on High Performance Computer Architecture*, Austin TX USA, 2 2006.

[7] B. C. C. J. J. G. P. H. K. K. D. A. P. W. L. P. J. S. S. W. W. Krste Asanovic, Ras Bodik and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 18 2006.

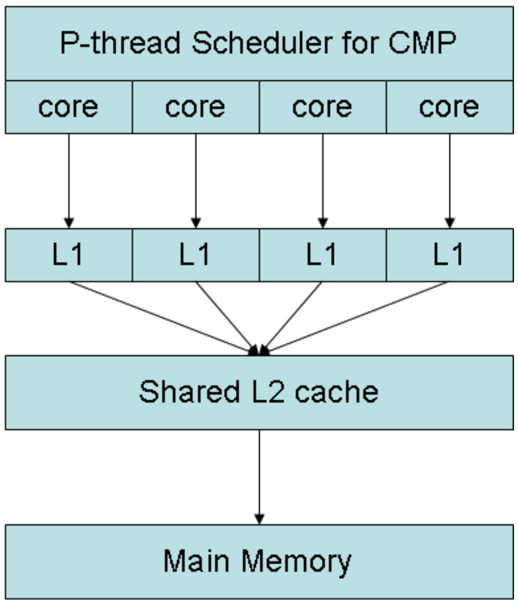

Figure 6: Simulated Architecture

[8] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM Conference on Programming Language Design and Implementation*, pages 190–200, Chicago, IL, June 2005.

[9] M. D. McCool, Z. Qin, and T. S. Popa. Shader metaprogramming. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 57–68, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[10] A. Phansalkar, A. Joshi, L. Eeckhout, and L. K. John. Measuring program similarity: Experiments with spec cpu benchmark suites. In *Proceedings of the 2005 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS05)*, pages 10–20, Mar. 2005.

[11] J. Robert H. Halstead. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.

[12] S. Rul, H. Vandierendonck, and K. De Bosschere. Function level parallelism driven by data dependencies. In N. P. Jouppi, R. Kumar, and D. M. Tullsen, editors, *Workshop on Design, Architecture and Simulation of Chip Multi-Processors (dasCMP)*, page 8, Orlando, FL USA, 12 2006.

[13] D. M. Tullsen and J. S. Seng. Storageless value prediction using prior register values. In *ISCA*, pages 270–279, 1999.

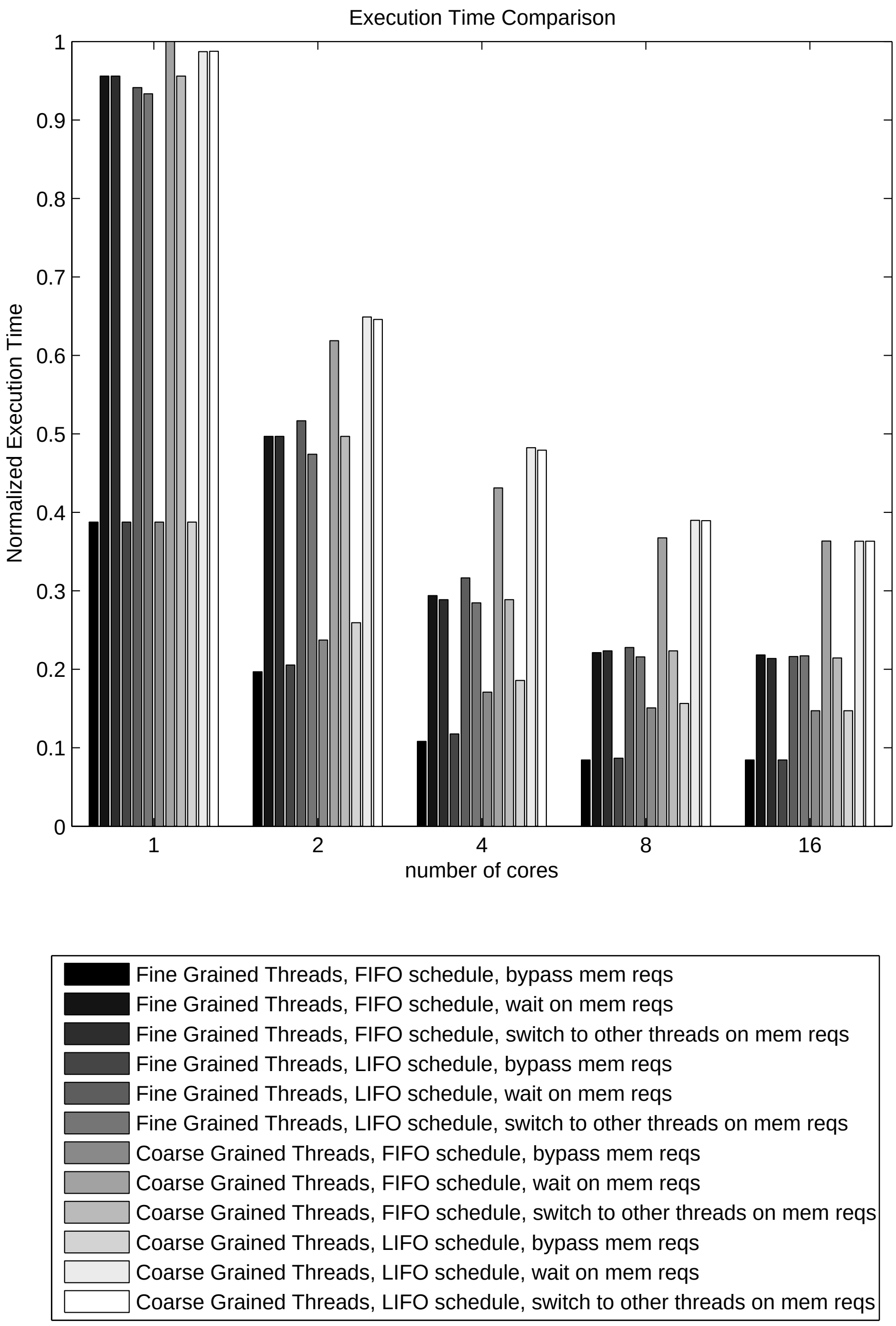

Figure 7: Execution time normalized to coarse grained threads running on one core with FIFO scheduling and they wait on every ...memory requests

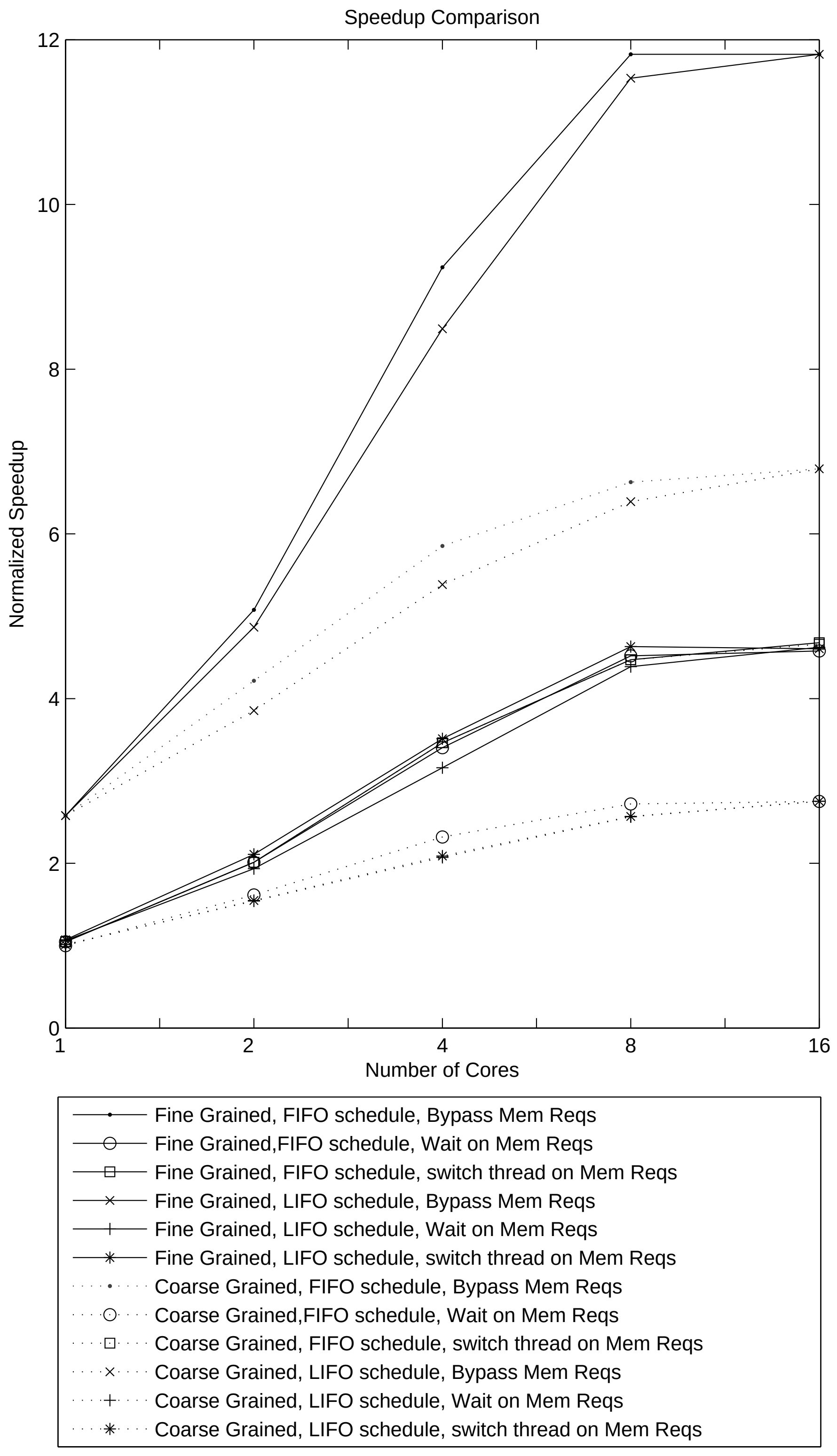

Figure 8: Speedup Normalized to coarse grained thread running on one core with FIFO scheduling and wait on memory requests

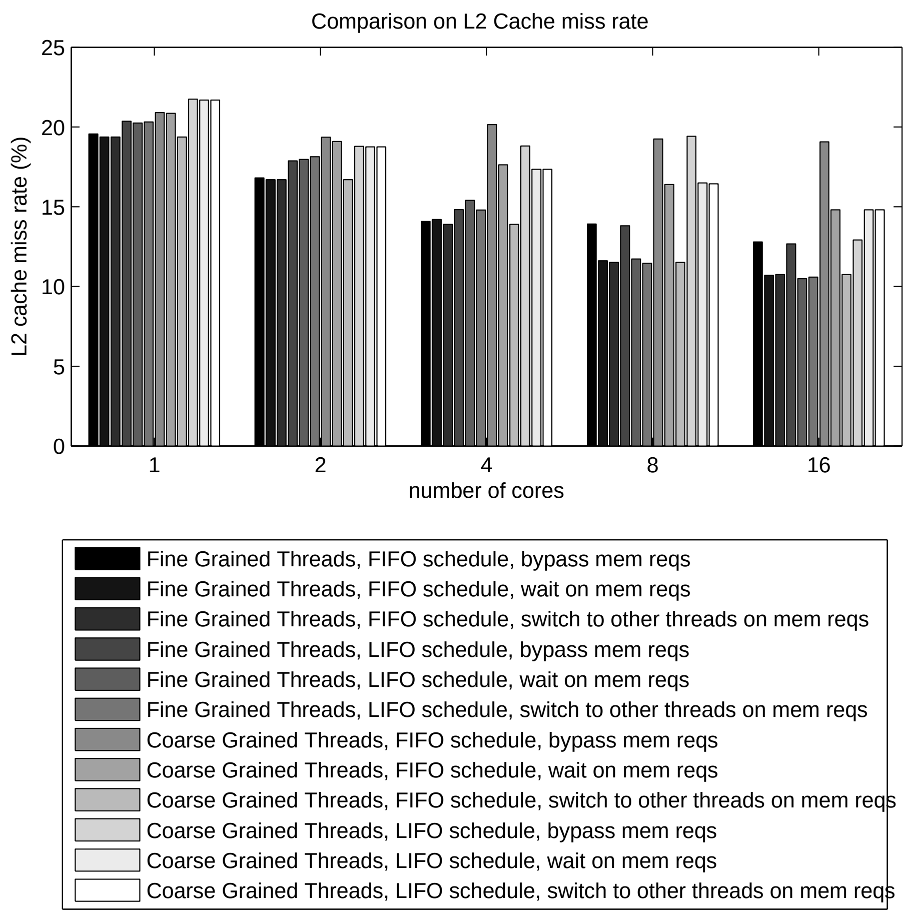

Figure 9: Comparing last level cache(L2) miss rate with different schemes.

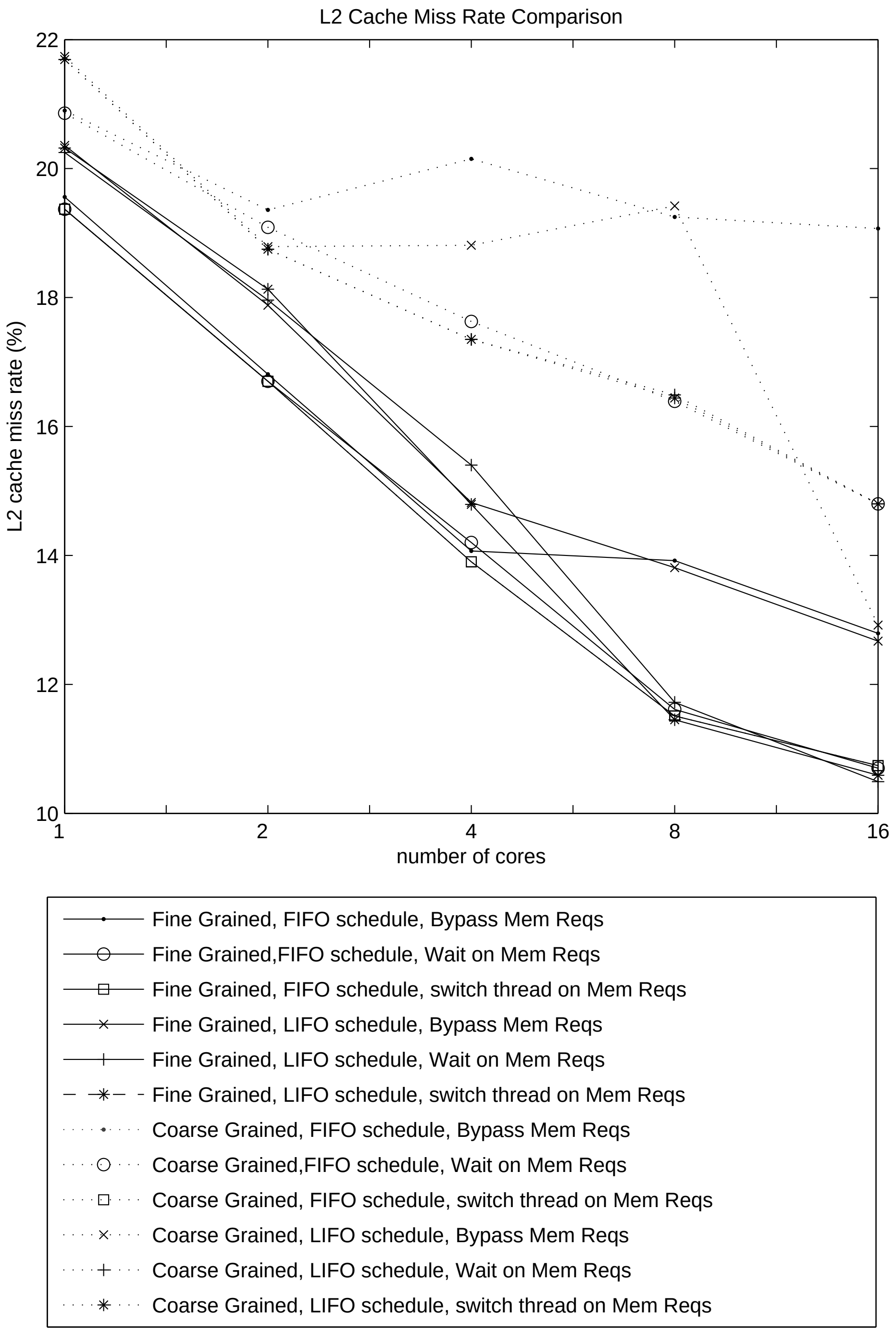

Figure 10: Comparing last level cache(L2) miss rate with different schemes.