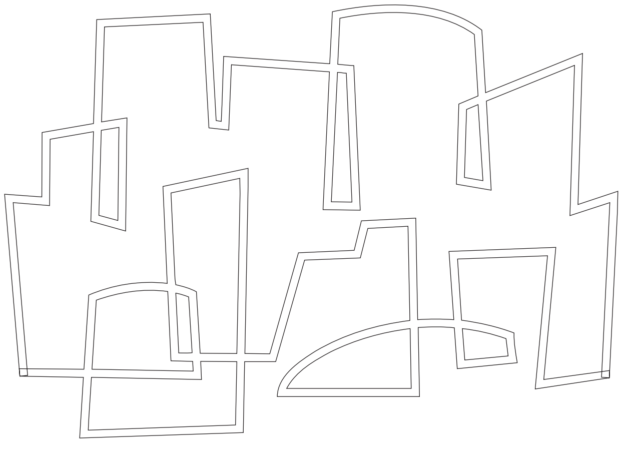
Local Memory Reference Behavior of Fine-Grain Multithreaded Execution

Masato Motomura, Gregory Papadopoulos

1992, November

Computation Structures Group
Memo 346



Local Memory Reference Behavior of Fine-Grain Multithreaded Execution

Computation Structures Group Memo 346
December 14, 1993

Masato Motomura
Gregory M. Papadopoulos

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-89-J-1988.

Local Memory Reference Behavior of Fine-Grain Multithreaded Execution

Masato Motomura[*]
Gregory M.Papadopoulos[†]

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

December 14, 1993

Abstract

Multithreading is a potentially important technique to deal with the effects of large communications latencies in multiprocessors. In this paper, we study the behavior of local memory references under a fine-grain multithreaded execution model. Based on the threaded abstract machine (TAM), the model is a compiling convention for distributed memory multiprocessors and does not presuppose special hardware support like multiple hardware contexts. We believe we have made three basic contributions to the understanding of local reference streams under fine-grained multithreading: (1) we extensively studied the execution traces from two example programs run on a multiprocessor simulator, (2) we have determined a power-law working set model for local activation frame and instruction references, and (3) using the working set model we have derived an analytic cache model which provides excellent agreement with acutal cache simulations.

Keywords: Multithreading, Cache Models, Working Set Behavior, Fine-Grain Parallelism, Latency Tolerance

[*]On leave from System ULSI Research Laboratory, Microelectronics Research Laboratories, NEC Corporation. Phone: 81-427-71-0709, Fax: 81-427-71-0881, Email: motomura@mel.cl.nec.co.jp

[†]Phone: 1-617-253-2623, Fax: 1-617-253-6652, email: greg@abp.lcs.mit.edu

1 Introduction

Multithreaded execution is often advocated as an effective latency-tolerance technique for multiprocessors. The term "multithreading" loosely applies to a fairly broad range of hardware mechanisms and compiling disciplines which share the common property of multiplexing, in time, several distinct computational contexts onto a single processor. The latency tolerance property is a consequence of the situation where at least one of the simultaneous contexts can make progress executing while some or all of the others might be blocked due to a long latency event, *e.g.,* a cache miss, explicit receive, or split-phase transaction.

Unfortunately, rapidly switching a processor among distinct contexts will, in general, decrease the locality of the processor's instruction, activation frame (stack frame), and heap reference streams with a commensurate increase in cache miss ratios [1, 2]. This study is concerned with quantifying, both analytically and experimentally, the effects on instruction and activation frame cache performance of an aggressive fine-grained multithreaded compiling model, the threaded abstract machine (TAM) model developed by Culler *et al* [3]. The instruction and activation frame reference streams comprise the *local* memory reference streams of TAM programs and typically are cached by processors as private or read-only data. Global references are made through explicit split-phase transactions, which can be thought of as a kind of binding prefetch into local memory (an entry in an activation frame). Here, we focus upon the locality properties of the local references.

In execution models like TAM, threads are short and non-blocking, and the number of potentially executing threads are not bound to fixed hardware resources such as the number of hardware contexts. Indeed, we make minimal assumptions about the underlying hardware, and in fact assume that the storage for, and scheduling of, multiple contexts are managed in software.[1]

We believe we have made three basic contributions to the understanding of local reference streams under fine-grained multithreading in the multiprocessor setting:

1. **Multiprocessor Execution Traces.** Using a multiprocessor instruction-level simulator, we have gathered extensive execution traces for two classes of applications: loop-parallel dense matrix arithmetic (matrix multiply – `MMP`) and procedure-parallel event simulations (Monte-Carlo neutral particle transport – `GAMTEB`). Previous studies obtained traces by artificially interleaving traces from several contexts [1, 2].
2. **Working Set Model.** Surprisingly, we have determined a working set model for local activation frame references that closely matches uniprocessor working sets for stack and heap data. The multithreaded working set is well-described by the power law function $W(t) = \alpha t^{\beta}$ — the size of the working set grows according to some power of the length of time over which references are considered. Similar to working sets reported for uniprocessors, the β for activation frame references is often around 0.5.
3. **Validated Analytic Cache Model.** Using the working set model for activation frame and instruction references, we have derived a set of useful formulae which predict cache performance under multithreaded execution. We have validated this model through actually simulating various cache configurations against the reference traces and have found excellent agreement.

The structure of the paper follows the items above. After quickly articulating the underlying multithreaded execution model, we describe the two codes we studied and the simulation environment. Careful study of the execution traces reveals the underlying power law working set model, which we then use as the basis for an analytic model of cache performance. Finally, we compare the model predictions with simulated caches and suggest possible scheduling techniques for improving miss ratios.

2 A Fine-Grain Multithreaded Execution

A number of approaches to multithreading for latency tolerance have been proposed and several have been implemented. Many schemes provide a substantial amount of hardware support for representing and scheduling the multiple contexts. The Denelcor HEP [9] provided cycle-by-cycle interleaving of threads which are each given their own hardware context of local registers. A thread which is blocking on a network transaction is not scheduled by a processor, and thus does not directly induce idle cycles, until the dependent transaction completes. Other researchers have proposed hardware support for a relatively few contexts for systems where global data caches can be relied upon to mask most of the high latency requests [1, 7].

Multithreading for latency tolerance can also be implemented primarily as a software technique with little or no special hardware support. The Threaded Abstract Machine (TAM) is a fine-grain threading model which evolved from the compiling of non-strict languages for hybrid dataflow/von Neumann machines [3]. Most TAM implementations are

[1]Practically speaking, however, fine-grained multithreading will only make sense on processors that support an efficient short message network interface. Otherwise network overhead will dominate execution time. Fortunately, tightly integrated network interfaces are appearing in a number of commercial and research machines, *e.g.,* the CM-5 [4], NCUBE, Tera [5], iWARP, J-Machine [6], Alewife [7] and *T [8]. Our results should give insight to the local memory reference behavior of codes running any of these machines which have been compiled for a TAM-like execution model.

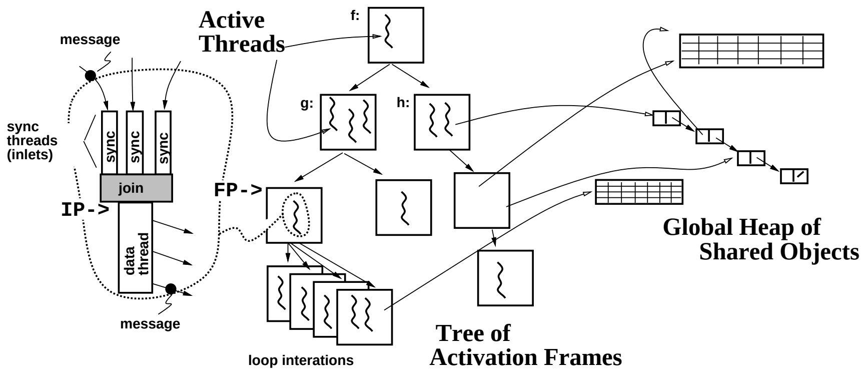


Figure 1: The execution state of a TAM program is a tree of activation frames. Each frame may, in general, have several concurrently executing threads. Work distribution is accomplished by giving each processor a subset of the frames, and thus the threads that correspond to them.

targeted to ordinary processors with no special support for multithreading, so this form of multithreading is rightly viewed as a compiling discipline where a thread's context is explicitly managed by the executing code.

Our study uses a TAM-like execution model on top of an otherwise conventional RISC augmented with an efficient user level interprocessor network interface.

2.1 Frames, Code Blocks and Threads

The execution state of a TAM computation comprises a *tree of activation frames,* one *frame* for each invoked *code block*, typically a procedure or a loop iteration. The frame provides the local storage for code block instances much like a stack frame in conventional sequential execution. Please refer to Figure 1. Statically, a code block consists of a set of interdependent *threads. Synchronization* (or *inlet*) threads are scheduled by the arrival of messages. Typically, a synchronization thread will copy a datum from the message into the frame and then perform a synchronization operation in the form of decrementing a counting semaphore in the current frame. If the synchronization condition is met (*i.e.,* the counter decrements to zero), then a *data thread* is queued for execution. A data thread is described by the simple continuation `(FP,IP)` where `FP` points to the current frame and `IP` points to the first instruction in the data thread. Data threads are processed whenever there are no incoming messages to process. That is, synchronization threads are preferred over data threads. The model is not preemptive; synchronization and data threads are nonblocking and, once scheduled, execute to completion. Also note that the synchronization threads can be directly mapped into so-called active message handlers (see [10]).

Within a frame, threads communicate with each other by sharing data in the frame, including the manipulation of counting semaphores (and potentially queueing another data thread for execution). Note that, without special optimizations, interthread communication does not occur through registers; each thread cold starts with an invalid register set (except for a register containing its `FP`). Communication between frames is strictly through messages. Also, operations against global storage (the heap) are accomplished by split-phase transactions. In a global read, for example, a thread emits a read request into the network and then continues computing (recall, threads cannot block). The remote memory processing responds with a message containing the desired location's value. This message is processed by a synchronization thread, just as if it had been directly communicated by a thread in some other frame.

Importantly, the execution model is *fully parallel.* Generally speaking, any of the frames in the tree can be responding to messages and have a supply of running or ready-to-run data threads. Work is distributed by giving each processor a subset of the frames, and thus the threads that correspond to them.

2.2 Loop Bounds

In the TAM model, each iteration of a loop can be given its own frame (various techniques of loop unrolling and frame reuse are employed to reduce the overhead of this). By allowing a number of iterations to exist simultaneously, loop iterations can proceed in parallel, limited only by the inherent data dependencies across iterations. By controlling the bound on the number of simultaneous loop iterations, a programmer can control the amount of parallelism exposed. *Loop bounds* can be specified dynamically on a loop-by-loop basis. We vary the loop bounds in our simulations in order to study the relationship between the amount of parallelism exposed and the locality in the frame and instruction reference streams.

2.3 Scheduling Policy

Another degree of freedom over the evolution of the program is the scheduling policy for ready-to-run data threads. In the original TAM work, each frame supported a local LIFO queue of enabled data threads. Then, all frames on a given processor with non-empty queues were linked together into a scheduling list. The scheduler always preferred data

Benchmark		GAMTEB	MMP
Program size (Inst.)		24,921	4,238
Number of Code Blocks		51	13
Data Threads	Avg. Number (/CB)	16.76	14.69
	Avg. Length (Inst.)	21.03	14.63
Sync. Threads	Avg. Number	28.31	19.69
	Avg. Length	4.81	5.64
Average Thread Length		10.84	9.48

Table 1: Static information of benchmark Programs. Program sizes and thread lengths are measured in 4-byte RISC-type instruction counts.

threads from the current frame, so a frame switch would occur whenever the local stack became empty or a message was processed which corresponded to another frame (and thus a synchronization thread was executed which corresponded to another frame). This policy provided very good performance on single processor implementations, but does not seem to have performed as well on multiprocessors because a single frame can not provide sufficient parallelism to mask communications latency.

In this study, we consider two very simple scheduling structures. On each processor, we have either a LIFO or FIFO queue of ready-to-run threads for that processor. In general, the queues will contain `(FP,IP)` pairs with many different `FPs`. That is, threads for different frames are mixed on the same scheduling stack. When a synchronization thread detects that its dependent data thread is enabled, the `(FP,IP)` pair corresponding to the data thread is enqueued. Whenever the scheduler requires more work, an `FP,IP` pair is dequeued and the scheduler jumps to the resulting `IP` to begin execution of the data thread.

3 Trace Driven Simulation

Previous simulation studies on the cache implications of multithreaded execution have used traces obtained by artificially interleaving reference streams from several contexts [1, 2] simply because real traces were not available.[2] Though this kind of approach has its meaning for the kind of context switching studied — essentially switching amongst a few (say, two or four) hardware contexts in order to mask cache misses — it is problematic for fine-grain multithreaded execution where strong data dependencies amongst threads have a first order effect upon scheduling and therefore working set behavior.

3.1 Benchmark Programs

For this study, we extensively explored the behavior of two distinctly different parallel programs: `GAMTEB`, a Monte Carlo neutral particle transport simulation of photons traversing through a carbon rod and `MMP`, an unblocked matrix multiply. The parallelism in `MMP` is obvious, while `GAMTEB` has one outermost loop comprising a set of recursive code blocks whose calling path depends on a number randomly generated for each iteration of the loop. Both programs were written in Id and compiled to TL0 code using the Berkeley backend [3]. While we certainly would desire to study more programs, we felt that these were largely representative of fairly broad classes of applications and preferred to apply our resources to study their behavior in a great deal of depth. Here is a brief description of the codes:

We ran the programs on various system configurations, input data sizes, and loop bounds. We report here simulation results on an eight processor configuration with 20,000 photons for `GAMTEB` and a 600 $\times$ 600 matrix for `MMP`. Three different loop bounds are experimented for each program: 50, 200 and 800 for `GAMTEB`, and 25, 50 and 100 for `MMP`. Note that loop bound in `MMP` is applied to all of the three loops, whereas the loop bound for `GAMTEB` is applied to the outer loop only. Furthermore, to see how scheduling influences memory reference behavior, we conducted experiments on thread scheduling policies: LIFO versus FIFO queues for enabled data threads.

3.2 Statistical Summary

Table 1 summarizes static information for these two benchmark programs. `MMP's` program size is about 16KB, which is apparently too small for the purpose of instruction cache simulation. For the frame cache, however, this program is a good test of locality behavior, especially considering that controlling the cache behavior of the matrices themselves can be quite tricky [13]. This is reflected in the frame references because the frames essentially hold pieces of the matrices which have been explicitly prefetched. In both programs, the average length of synchronization threads (called inlets in the TAM model), around five instructions, are considerably shorter than those of data threads, which are well above ten instructions.

Table 2 shows trace length for two runs out of the twelve runs that we examine in this paper. Other runs for each

[2]Note that others have performed realistic simulation of multithreaded codes, however these do not seem to account for local cache behavior. For examples, see [11] and [12].

Bench-mark	Sched. Policy	Loop Bnd	Num. of Threads	Inst. Refs.	Frame Refs.
`GAMTEB`	LIFO	200	0.69M	7.95M	3.66M
`MMP`	LIFO	50	0.81M	9.40M	4.60M

Table 2: Trace length examples.

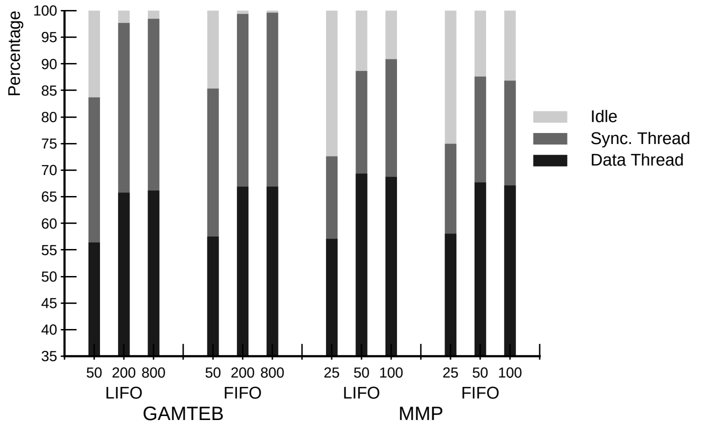


Figure 2: Processor utilization for 12 runs. The processor time is divided into Idle time, Synchronization thread execution time and Data thread execution time. Note that the y-axis minimum is not zero.

benchmark have essentially the same characteristics. Rather than record the execution traces from all eight processors, we instead recorded references from only one of the processors until the statistics we report later reached a stable state and the traces are long enough for simulations of caches smaller than 64KB. For the purpose of justifying using traces for one node, we verified that statistics variations between nodes were negligible.

Figure 2 shows processor utilization of each run by dividing processor time into data thread execution time, synchronization thread execution time and idle time. The numbers 50, 200, 800 and 25, 50, 800 refer to the loop bounds used in `GAMTEB` and `MMP`, respectively. `GAMTEB` achieves near perfect utilization with loop bounds of 200 and greater. `MMP` only achieves a maximum of 85% utilization, which we attribute to the fact that the unblocked algorithm exhausts the interprocessor communications bandwidth. Observe that there isn't significant difference in processor utilization between LIFO and FIFO scheduling, though slight improvement can be found for `GAMTEB`. We caution that Figure 2 should not be thought as predicting performance of these programs; this is not our objective. Rather, the important consequence of this figure is that choice of our loop bounds are just around a saturation point of processor utilization for the duration of program execution we study. The saturation point is a point where processor resource usage is optimized [14].

4 Working Set Behavior

Central to the prediction of cache behavior is the characterization of temporal correlations in reference streams. A *working set* $W(t)$ [15] is a function of a time window t which describes the number of unique references that occur in the window. If $W(t)$ grows slowly with t then the reference stream has good locality. Often times, working sets will tend to follow a power law,

$$W(t) = \alpha t^{\beta}. \tag{1}$$

Typically, values for β around 0.5 are encountered in conventional architectures [16]. That is, the working set grows like the square root of the time window size considered. Observe, if $\beta \approx 1$ then the reference stream is devoid of locality. We have found that, for frame references in `GAMTEB`, β ranges from 0.45 to 0.71, and thus exhibit fair amount of locality. Unfortunately (and surprisingly), instruction references in `GAMTEB` experience very poor working set behavior ($\beta = 0.99$) until the window grows large enough to encompass all of the code blocks included in the main outer loop. `MMP` generally experiences the same frame reference locality before its working set function shows saturation, but has excellent instruction locality due to its small inner loop. In this section we detail our technique for identifying the working set behavior for frame and for instruction references.

4.1 Frame Reference Patterns

Table 3 shows frame reference statistics for each run. The dynamic average frame size is reported in double words (eight bytes), and is about 80 double words for `GAMTEB` and 20 double words for `MMP`. The small variation of frame sizes among runs for each program reflects the different path each run takes depending on a loop bound and a scheduling policy. Frame lifetime, which is the time from an allocation to a deallocation of a frame, is measured as a total number of executed threads in a processor. As expected, frame lifetime strongly depends on both scheduling policy and loop

Benchmark	Scheduling	Loop Bnd	Frame Size	Frame Lifetime (1000's)	Frame Execution Time	Number of Extant Frames	Context Switch Interval
GAMTEB	LIFO	50	80.25	2.98	62.92	63.38	2.70
		200	80.86	5.29	62.94	161.49	2.86
		800	81.61	6.56	64.06	372.96	2.65
	FIFO	50	81.21	1.85	63.91	39.51	1.86
		200	79.06	8.04	62.25	169.95	1.40
		800	80.47	15.45	62.27	374.81	1.41
MMP	LIFO	25	17.45	11.31	2671.69	18.21	2.24
		50	19.14	20.02	2469.50	43.01	2.00
		100	20.30	32.03	2339.11	92.25	2.10
	FIFO	25	17.24	12.31	2512.49	21.86	1.48
		50	18.45	22.78	2461.87	43.12	1.14
		100	19.53	46.45	2368.63	89.93	1.13

Table 3: Frame Statistics. All numbers are average. Frame size is measured in 8-Byte words. A number of executed threads is used as a measure of time. A difference of length between data thread and synchronization thread is ignored.

bounds. Frame execution time is measured in threads and denotes the average number of threads executed in a given frame during its lifetime. The number of extant frames describes the average number of frames which exist at any given moment in time. Note how this value is directly proportional to the loop bound, but is relatively independent of scheduling policy. Context switch interval, again measured in threads, describes the average number of threads executed in a given frame before executing a thread for another frame. Here, a LIFO scheduling apparently yields somewhat longer context switch intervals.

There is a significant difference between GAMTEB and MMP in frame execution time to lifetime ratio. Comparatively, MMP is working on smaller frames for much longer time than GAMTEB, which implies better frame cache hit ratio for MMP. Paradoxically, the context switch interval is *longer* in GAMTEB — surprisingly so, given that other statistics shows better locality for MMP. This means simple view of treating context switch interval as a measure of locality [1, 2] is not a good predictor for fine-grain multithreading. We need to find a more reliable metric.

4.2 The Frame Working Set Function

We define a *frame working set function*, $W_F(t)$, as a number of distinctive frames on which threads are executed during time window t, where t is measured by a dynamic count of threads.

Figure 3 shows measured $W_F(t)$ for all of runs. LIFO scheduling for GAMTEB shows power law behavior, which is independent of loop bounds. MMP's frame working set function shows saturation after initial rise whose slope is compatible with GAMTEB. Intuitively, the power law for GAMTEB implies that this program's working set has a "transient" nature: some portion of a working set is continuously changing along with the execution of a program, which results in a continuously growing frame working set function. In contrast, the saturation behavior for MMP means that this program's working set has a "recurrent" nature: the working set is closed in the sense that same fixed number of frames are used over a long interval of time.

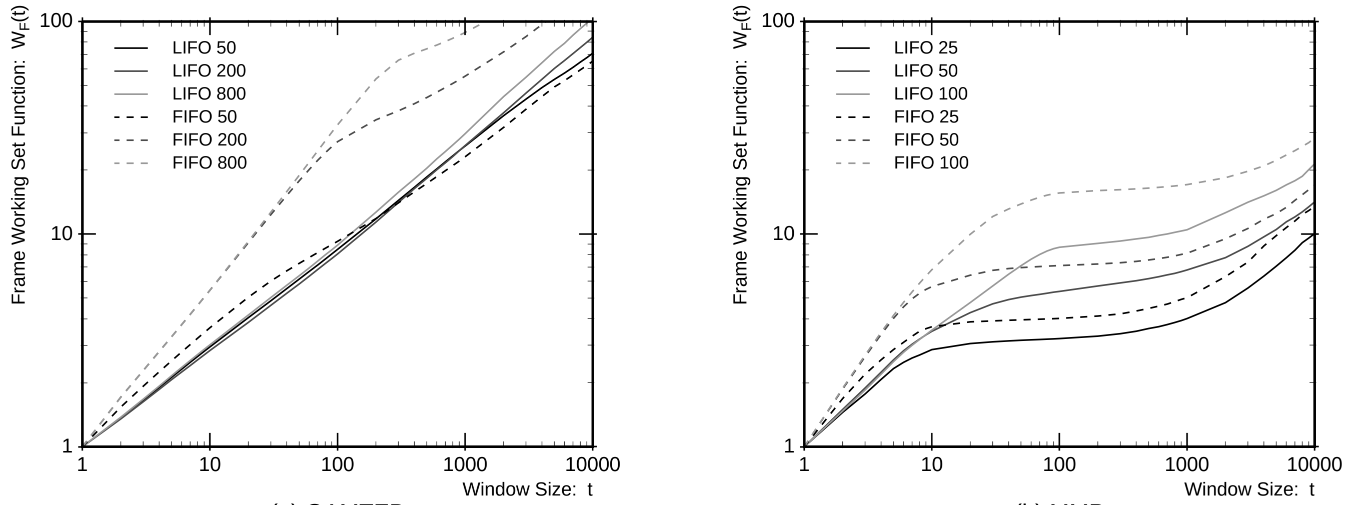
The range of power constants for GAMTEB, from 0.45 to 0.71, overlap the values found in conventional architectures, which range from 0.484 to 0.544 [16], and from 0.43 to 0.75 [17]. This suggests that the amount of locality in frame memory references may be similar to memory references in conventional architectures at least for LIFO scheduling. This is a nice result if holds true for most multithreaded applications because it suggests that the local memory systems designed for cacheing stack and heap references on sequential processors could support the demands of frame cacheing.

4.3 The Instruction Working Set Function

Unfortunately, a general framework for instruction reference locality is more complicated than for frames. Because threads from different frames may point to the same code, but threads from the same frame cannot (lest they not be unique), we have to separately investigate how code blocks and threads are scheduled. As such, we report measurements of two functions, *code block working set function* $W_B(t)$ and *thread working set function*, $W_T(t)$. Those are defined as distinct numbers of code blocks or threads referenced during a given period of time t, respectively.

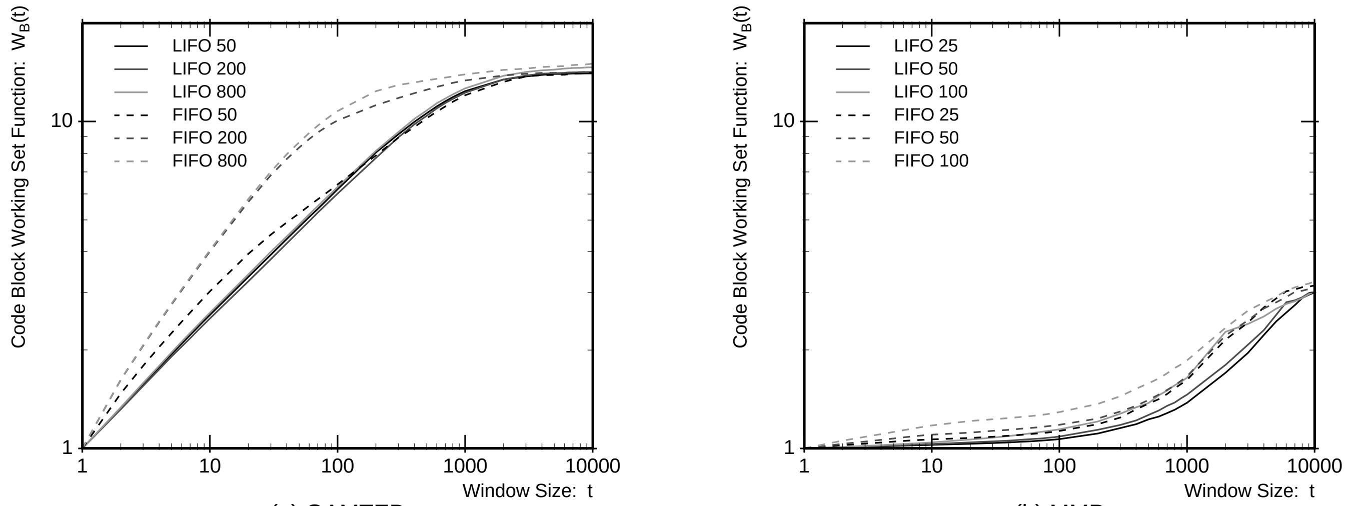
Figure 4 shows measured $W_B(t)$. Since the number of code blocks is limited, both of the curves show saturation. GAMTEB shows again power law behavior with β ranging from 0.42 to 0.56. In MMP, the working set is limited almost to a single code block, which is the code block of the inner-most loop.

Figure 5 shows measured $W_T(t)$ which shows how temporal locality in instruction references is exhibited for each program. Here, GAMTEB has very poor temporal locality; β is 0.99, which means hardly any thread is re-executed during



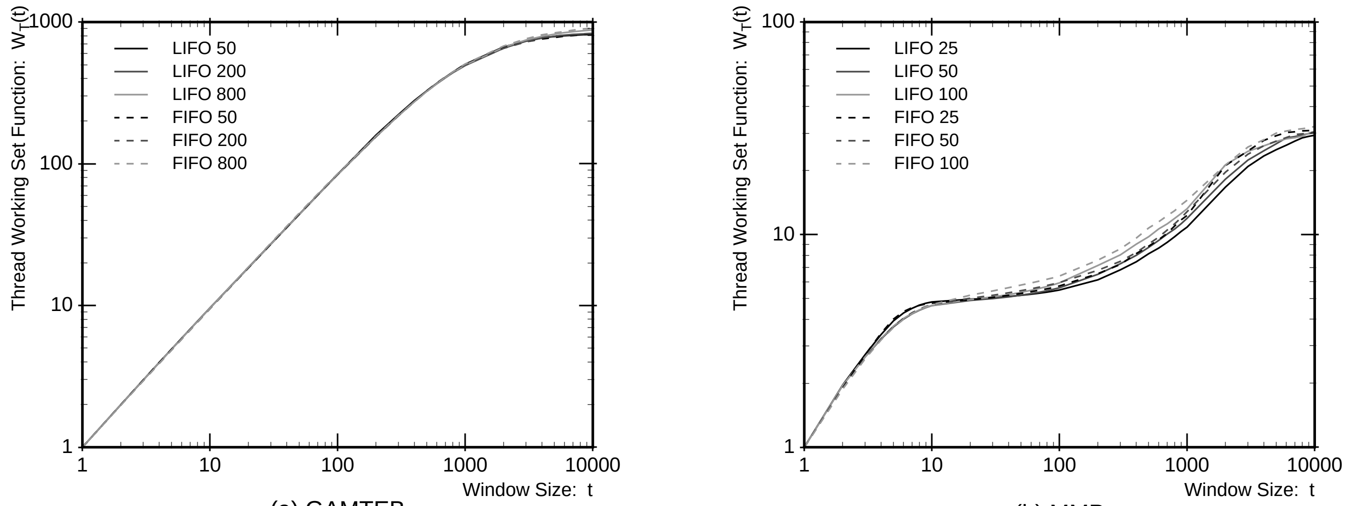
(a) GAMTEB (b) MMP

Figure 3: Measured frame working set function $W_F(t)$ for (a) GAMTEB and (b) MMP. This function shows how many new frames are scheduled for execution within a given period of time window. Time t is measured by dynamically executed number of threads.



(a) GAMTEB (b) MMP

Figure 4: Measured code block working set function $W_B(t)$ for (a) GAMTEB and (b) MMP. This function shows how many new code blocks are scheduled for execution within a given period of time window.



(a) GAMTEB (b) MMP

Figure 5: Measured thread working set function $W_T(t)$ for (a) GAMTEB and (b) MMP. This function shows how many new threads are scheduled for execution within a given time window.

this period, regardless of scheduling and loop bounds. This is a disappointing but still an understandable result because GAMTEB has only one outermost loop and the execution pass inside of the loop varies on each iteration. In MMP, though LIFO seems to maintain better overall temporal locality, FIFO performs slightly better in small working set region before reaching a knee around size of 5 threads.

5 An Analytic Cache Model

In the previous section, we have shown how a working set function can capture the locality behavior of both frame and instruction references quite well. In this section, we will build an analytic cache model for a fine-grain multithreaded execution. The model predicts measured cache miss ratio quite well, as we will see in a next section. Though our cache model is geared toward explaining cache behavior for multithreading, it incorporates what we believe to be a new and general idea for using working set function as a basis for calculating cache miss ratios. This idea is readily applicable to any cache modeling wherever an underlying working set model applies.

5.1 Previous Work in Analytic Cache Modeling

The most important and most extensive work in modeling cache behavior was done by Agarwal, Hennesy and Horowitz [18]. This work proposed a full framework for understanding cache behavior for various cache configurations including differing line sizes and associativity. Though this work gives us good insight into cache behavior, we believe several points are yet to be solved by this model.

First of all, a time granule, which they used for measuring a working set size, was chosen in a somewhat ad hoc manner. Although it is argued that the choice of time granule does not have a significant effect on the model's prediction accuracy, we believe there should be some systematic way of deciding this parameter. Secondly, a parameter called the collision rate, which is measured from traces separately, plays a key roll in their modeling. Since a working set model seems to work as well for fine-grain multithreaded execution as it does for sequential execution, we would like parameters to be derived directly from a working set function. Lastly, an LRU replacement strategy, which has practical importance for set associative cache is not modeled in their work.

Thiebaut has also conducted important work in this area [16] centered around the notion of fractal random walks. In fact, the power law observed in the working set functions can be explained in terms of the self-similarity of fractal random walks. Unfortunately, Thiebaut limits his analytic model to fully associative caches.

In a context of multithreaded architecture, we are aware of only the work of Agarwal [2], which is a direct descendent of work described in [18]. Agarwal treats multithreading as an extreme case of multiprogramming, where context switch interval is not long enough to fully bring its working set into cache. The number of contexts is treated as an artificially controllable parameter. As we have stated, this treatment seems to be insufficient for fine-grain multithreading wherein data dependencies amongst threads have a first order effect upon scheduling and thus cache behavior.

5.2 Working Sets for Cache Modeling

A working set is a monotonically increasing function of time. Thus, the size of a working set for a program depends on the size of the time window, τ, selected for that working set. In other words, in an intuitive notion of working set, *i.e.,* where a program is "currently" focusing its computation, a choice of the window size implied by "currently" changes the working set size completely. This is a fundamental difficulty of modeling cache behavior based on a notion of working sets.

We claim that τ should depend on cache size. Caches of different size "see" memory reference streams with different "time resolution". We must account for the different time resolution in order to coherently model different sized caches.

We introduce *working set growth ratio* p as,

$$p = \tau \frac{W'(\tau)}{W(\tau),} \tag{2}$$

where $W(\tau)$ is the working set function at time τ, a working set size, and $W'(\tau)$ is a time t derivative of $W(\tau)$. p is the ratio of the newly referenced working set within $W(\tau)$. That is, $(1-p)W(\tau)$ is carried over from a working set in an adjacent τ period. For example, if p is 0, the working set function is not growing at all. If p is 1, there is no overlap in adjacent τ period, which means the working set is increasing linearly against time. We can derive a general equation which $W(\tau)$ should satisfy.

$$W(\tau) \ \times \ \textit{Hit Ratio} \ = r(p) \ \times \ \textit{Cache Capacity}, \tag{3}$$

where $r(p)$ is an yet unknown function which will be defined below. The left hand side of Equation (3) gives the size of the cache resident portion of a working set, since the hit ratio indicates the percent of a working set that should be cache resident in order to insure a cache hit. Thus, Equation (3) simply says the cache resident portion of a current working set occupies $r(p)$ of the cache capacity. $(1-r(p))$ of a cache is left over from references which have drifted out of the current working set. There are two conditions that $r(p)$ should satisfy. $r(0)$ should be equal to 1 because the working set size is not increasing when $p=1$. $r(1)$ should be equal to 0 because no memory location is being reused, the left hand side of the equation is 0. $r(p)$ can be chosen arbitrarily provided that it satisfies these two boundary conditions. Thus we can simply set

$$r(p) = 1 - p. \tag{4}$$

Since the miss ratio of a cache is calculated using $W(\tau)$, Equation (3) is actually a self-consistency equation that $W(\tau)$ and a miss ratio should satisfy. This is why we could choose $r(p)$ freely under two boundary conditions.

Mathematically, a miss ratio is obtained as a fixed point of self-consistent equations, which we will derive following sections. This self-consistency requirement is the key to solve the *chicken and egg* problem of determining the working set size. This technique is applicable to choosing the working set size required to model any cache.

We can now see why the previous study that chose τ arbitrarily [18] was successful. Suppose power law applies to a working set function, *i.e.*, $W(t) = \alpha t^{\beta}$. The working set growth ratio is calculated as,

$$p = \tau \frac{\alpha \beta \tau^{\beta - 1}}{\alpha \tau^{\beta}} = \beta. \tag{5}$$

This ratio does not depend on τ. It means, as we will see, that miss ratios do not depend on τ either. This is one example of the self-similar behaviors generally observed in fractal random walks. The simple power law does not apply to all cases, however. In many cases, there are often changes to the power constant as the time window grows [19]. Moreover, the working set function eventually shows saturation. The working set size's dependency on the cache size must be introduced to predict cache miss ratio for these cases correctly.

5.3 Frame Cache

Now we apply the general idea from the previous section to model an activation frame cache behavior. Several mean values that characterize frame reference behavior as well as a measured frame working set function are used for the model. We make several assumptions. First, we assume there is no self-interference on a cache set among references that belong to a same frame. Secondly, we assume random mapping of `FPs` on cache lines, and random memory references within a frame. We model only the *warm start miss ratio*, because we are modeling a multithreaded execution model in a single programming environment.

We calculate the miss ratio of a frame cache by counting how many references from a thread miss on average. Let T_F be an average number of total frame references from a thread, and M_F be an average number of missed references in a thread. A *frame cache miss ratio* m_F is defined as

$$m_F(L, A, S) = \frac{M_F(L, A, S)}{T_F}. \tag{6}$$

Here, we use L for a line length measured in bytes, A for associativity in a set, and S for number of sets, so that cache capacity is expressed as LAS. In the following discussion, we omit L, A, S dependencies wherever appropriate.

T_F can be divided into three components: T_{Fc}, the average number of first references to cache lines of a frame during that frame allocation, the T_{Ft}, average number of references to cache lines which were referenced by other instructions in a same thread, and the T_{Fi}, average number of references to cache lines which were not referenced in a same thread.

Generally, each of the T_{Fc} references produces a miss, which is known as a *compulsory miss*, because it is the first access to a line in any way. It doesn't hold for our case, however, because T_{Fc} is a first request to frame locations within one allocation of a frame, and a same `FP` might have been used before the current allocation. This possible previous allocation might have left several lines in a cache which reduce compulsory misses. Since a frame is a local storage of a code block invocation, a compiler guarantees that there is no data consistency problem between different frame allocations. Thus, we can define compulsory miss ratio m_{Fc}, so that $m_{Fc}T_{Fc}$ is a number of compulsory miss references.

T_{Ft} is a measure of temporal and spatial locality within one thread. It is independent of dynamic execution of threads, because a thread is guaranteed to execute until its completion. Each of T_{Ft} references gives a hit. To get T_{Ft} from a compiled code is quite straightforward.

T_{Fi} is a most important source of misses, which are known as *interference misses.* We define m_{Fi} as an *interference miss ratio*, so that $m_{Fi}T_{Fi}$ gives a number of interference miss references. In total, a frame cache miss ratio is given by

$$m_F = \frac{T_{Fc}}{T_F} m_{Fc} + \frac{T_F - T_{Fc} - T_{Ft}}{T_F} m_{Fi}. \tag{7}$$

Direct Map Cache Case

We first show the calculation of m_{Fi} for a direct map cache. Let F be an average frame size measured in bytes. The number of cache lines required to hold a frame size of F is calculated by using the *cover function C* [18] as

$$C(F) = 1 + \frac{F - 1}{L}. \tag{8}$$

Suppose, on average, R_F lines of a frame which is contained in a working set are resident in a cache. Based on an idea in Section 5.2, the working set size for frame references is defined as a frame working set function measured at some time interval τ, $W_F(\tau)$. By definition of this function, the working set size means the number of frames that constitute a working set. Thus, Equation (3) is dictated into a following equation:

$$W_F(\tau) R_F = (1 - p_F) S, \tag{9}$$

because $W_F(\tau)R_F$ gives the number of cache resident lines of the working set. Here, the *frame working set growth ratio* p_F is given by

$$p_F = \tau \frac{W_F'(\tau)}{W_F(\tau)}. \tag{10}$$

Since the average fraction of a frame that is not resident in a cache should give an average miss ratio,

$$m_{Fi} = 1 - \frac{R_F}{C(F).} \tag{11}$$

Thus Equation (9) gives a frame working set size as a function of m_{Fi}.

$$W_F(\tau) = \frac{1-p_F}{1-m_{Fi}} \frac{S}{C(F)}. \tag{12}$$

As explained in Section 5.2, Equation (12) is a self-consistency equation that m_{Fi} and $W_F(\tau)$ should satisfy.

Now, we want to derive an equation which gives R_F. Look at one thread to see which frame it belongs to. If it belongs to a frame within a working set, R_F will increase because of addition of new lines. If not, R_F will decrease because of interference in a cache. By requiring these increment and decrement for R_F to be balance at a steady state, following equation is obtained.

$$T_{Fi}(1-p_F)\left(1 - \frac{R_F}{C(F)}\right) = T_{Fi} p_F \frac{R_F}{S}. \tag{13}$$

Here, $\left(1 - \frac{R_F}{C(F)}\right)$ gives a cache non-resident fraction of a frame in a working set. Since there are T_{Fi} references from a thread, the increment of R_F in Poisson approximation is given by the left hand side of the equation. The right hand side gives a decrement of R_F, because $\frac{R_F}{S}$ gives a probability that interference occurs. Thus,

$$R_F = \frac{C(F)}{1 + \frac{p_F}{1-p_F}\frac{C(F)}{S}}. \tag{14}$$

is obtained. m_{Fi} is easily derived from R_F by using Equation (11) as

$$m_{Fi} = \frac{1}{1 + \frac{1-p_F}{p_F}\frac{S}{C(F)}}. \tag{15}$$

It turns out that p_F and $\frac{S}{C(F)}$ are two important parameters that completely decide this miss ratio. If $p_F = 1$, $m_{Fi} = 1$ as expected.

Calculation of compulsory miss ratio m_{Fc} can be carried out in a similar manner. Let R_c be an average number of cache lines which hold a frame when that frame is deallocated. Suppose the probability of reuse of the deallocated frame is p_r. The steady state equation for R_c is given by

$$C(F)p_r\left(1 - \frac{R_c}{C(F)}\right) = C(F)(1-p_r)\frac{R_c}{S} + C(F)(N_c - 1)\frac{R_c}{S}. \tag{16}$$

The left hand side and the first term of the right hand side can be understood in a same fashion as corresponding terms in Equation (13). These terms are multiplied by $C(F)$ because $C(F)$ distinct lines are referenced within a frame allocation. The second term of right hand side is turbulence from other concurrently executing frames. Here, N_c is the average number of frames in a working set during frame life time interval τ_L. Thus,

$$N_c = \frac{1}{\tau_L}\sum_{t=1}^{\tau_L} W_F(t). \tag{17}$$

From Equation (16),

$$R_c = \frac{C(F)}{1 + \frac{N_c - p_r}{p_r}\frac{C(F)}{S}} \tag{18}$$

is obtained. m_{Fc} is given by R_c as follows,

$$\begin{aligned} m_{Fc} &= p_r\left(1 - \frac{R_c}{C(F)}\right) + (1-p_r) \\ &= 1 - \frac{p_r}{1 + \frac{N_c - p_r}{p_r}\frac{C(F)}{S}}, \end{aligned} \tag{19}$$

because all of T_{Fc} references will miss a cache if a frame is allocated for the first time.

In summary, we get a complete frame cache miss ratio for direct map cache as follows:

$$\begin{aligned} m_F &= \frac{T_{Fc}}{T_F}\left(1 - \frac{p_r}{1 + \frac{N_c - p_r}{p_r}\frac{C(F)}{S}}\right) \\ &\quad + \left(1 - \frac{T_{Fc}}{T_F} - \frac{T_{Ft}}{T_F}\right)\left(\frac{1}{1 + \frac{1-p_F}{p_F}\frac{S}{C(F)}}\right). \end{aligned} \tag{20}$$

For this model, we need the following average numbers and a function: the number of frame memory references per thread T_F, the number of compulsory references T_{Fc}, number of frame references that are covered within a thread T_{Ft}, the size of a frame F, the frame life time τ_L, and a frame working set function $W_F(t)$. Note that these parameters and function does not depend on any cache parameter. Though T_{Ft} depends on line size, this dependency is calculated by using a cover function C. T_F and T_{Ft} is obtained by static analysis. Moreover, it is also possible to get a good estimate of T_{Fc} and F statically. Thus only τ_L and $W_F(t)$ are the dynamic components of our modeling.

Set Associative Cache Case

We model an LRU set associative cache by again neglecting self-interference on a cache set. Though it was a fairly good assumption for direct map caches, it is not necessarily a good assumption for set associative caches, because S decreases as A increases for a same size of cache. We use this assumption, however, because it gives us good insight into the cache behavior simply. Self-interference can be introduced into a model quite easily, but is more complex. Note this assumption gives an optimistic miss ratio when inappropriately applied.

First of all, a self-consistency equation for the frame working set in this case is given as,

$$W_F(\tau) = \frac{(1-p_F)}{1-m_{Fi}} \frac{AS}{C(F)}. \tag{21}$$

we divide R_F into $R_F(1)$ to $R_F(A)$, depending on the position in an LRU stack of a corresponding line. For example, $R_F(1)$ is a number of cache resident lines marked most recently used. We solve following steady state equations for each component of R_F.

$$T_{Fi}(1-p_F)\left(1-\frac{R_F(1)}{C(F)}\right) = T_{Fi}p_F\frac{R_F(1)}{S}, \tag{22}$$

$$T_{Fi}p_F\frac{R_F(i-1)}{S} = T_{Fi}p_F\frac{R_F(i)}{S} + T_{Fi}(1-p_F)\frac{R_F(i)}{C(F)}, \tag{23}$$

where $1 < i \leq A$. The left hand sides give increments of R_F's, while the right hand sides give decrements. Equation (22) is basically the same equation as Equation (13). In Equation (23), increment is due to probability of interference on a set in $R_F(i-1)$, which causes a line to shift its position in an LRU stack from $i-1$ to i. Decrement is due to same interference on $R_F(i)$, a first term, and a reference to a line in $R_F(i)$ itself which makes that line be brought into top of an LRU stack. Equations (22) and (23) are easily solved as,

$$R_F(1) = \frac{C(F)}{1+\frac{p_F}{1-p_F}\frac{C(F)}{S}}, \tag{24}$$

$$R_F(i) = \left(1-\frac{R_F(1)}{C(F)}\right) R_F(i-1). \tag{25}$$

Since R_F is given by a sum of this series, m_{Fi} is obtained by using Equation (11) as, simply,

$$m_{Fi}(L,A,S) = m_{Fi}^A(L,1,S), \tag{26}$$

which means m_{Fi} for a cache of set associativity A is given by a power A of m_{Fi} of a direct map cache which has S lines.

The compulsory miss ratio can be calculated in a similar manner. The result is

$$m_{Fc}(L,A,S) = m_{Fc}^A(L,1,S). \tag{27}$$

5.4 Instruction Cache

An instruction cache needs small changes in modeling because of two kinds of locality components that should be considered separately (see Section 4.3). We will show how two kinds of working set functions, code block and thread, are incorporated into instruction cache modeling.

Let T_I be an average number of instruction reference in a thread, i.e., thread length. We will calculate *instruction cache miss ratio* m_I again by counting how many references miss a cache out of T_I references. We neglect compulsory misses in an instruction cache in order to simplify the notation, since compulsory misses are a negligible component in any practical program. Since a thread occupies consecutive T_I locations in an instruction memory, the number of cache lines which cover one thread is given by $C(T_I)$. Thus, m_I is calculated as,

$$m_I = \frac{C(T_I)}{T_I} m_{Ii}, \tag{28}$$

where m_{Ii} is an interference miss ratio for an instruction cache.

We need some preparation in order to calculate m_{Ii}. First $C(T_I)$ is divided into two components: one is an average number of lines that are shared by two or more threads, $C_B(T_I)$, and the other is an average number of lines that belong only to one thread, $C_T(T_I)$. These are derived as follows.

$$C_T(T_I) = \frac{1}{N_T} C(B), \tag{29}$$

$$C_B(T_I) = \frac{1}{N_T} \left\{ N_T C(T_I) - C(B) \right\}, \tag{30}$$

where N_T is the average number of threads in a code blocks, and B is the average size of a code block. This division of $C(T_I)$ is quite important in order to model an instruction cache behavior correctly. A cache line in $C_T(T_I)$ is never brought into a cache unless a thread which occupies that line is executed, whereas a line in $C_0(T_I)$ can be brought by several threads in a code block. In other words, the locality that helps increasing cache residency of $C_T(T_I)$ is not of code blocks but of threads. On the other hand, code block level locality plays an important role for $C_B(T_I)$. Thus, $C_B(T_I)$ and $C_T(T_I)$ should be treated in terms of how many distinct code blocks or threads have been executed, respectively. This is why we need two kinds of working set functions.

Now we define cache resident portion of $C_B(T_I)$ and $C_T(T_I)$ as R_B and R_T, respectively. Since $C_T(T_I)$ and $C_B(T_I)$ does not overlap each other, a self- consistency equation to give working set size and its measurement time τ is

$$\frac{W_B(\tau) N_T R_B}{1 - p_B} + \frac{W_T(\tau) R_T}{1 - p_T} = S. \tag{31}$$

Here, each of *working set growth ratios* p_B and p_T is defined as

$$p_B = \tau \frac{W_B'(\tau)}{W_B(\tau)}, \qquad p_T = \tau \frac{W_T'(\tau)}{W_T(\tau)}. \tag{32}$$

As in frame cache modeling, we count increment and decrement for R_B and R_T when a thread is executed. Steady state equations for R_B and R_T are respectively given by

$$(1 - p_B) \frac{C_B(T_I)}{C(B)} (C_B(T_I) - R_B) = p_B C(T_I) \frac{R_B}{S}, \tag{33}$$

$$(1 - p_T)(C_T(T_I) - R_T) = p_B C(T_I) \frac{R_T}{S}. \tag{34}$$

In a left side of Equation (33), $\frac{C_B(T_I)}{C(B)}$ gives a probability that a thread references the $C_B(T_I)$ portion of a code block. If it does, cache resident portion increases from R_B to $C_B(T_I)$. The right hand side of the same equation is multiplied by $C(T_I)$ because any reference from a thread which belongs to a code block outside of the working set may cause interference. In Equation (34), the right hand side is multiplied by p_B not p_T, because a thread which belongs to a code block working set does not cause interference. These equations give

$$R_B = \frac{C_B(T_I)}{1 + \frac{1-p_B}{p_B} \frac{C(B)}{C_B(T_I)} \frac{C(T_I)}{S}}, \tag{35}$$

$$R_T = \frac{C_T(T_I)}{1 + \frac{1-p_B}{p_T} \frac{C(T_I)}{S}}. \tag{36}$$

Thus an instruction cache miss ratio m_I is given as

$$\begin{aligned} m_I &= \frac{C(T_I)}{T_I} \left(1 - \frac{R_B + R_I}{C(T_I)} \right) \\ &= \frac{C_B(T_I)}{T_I} \frac{1}{1 + \frac{p_B}{1-p_B} \frac{C_B(T_I)}{C(B)} \frac{S}{C(T_I)}} \\ &\quad + \frac{C_T(T_I)}{T_I} \frac{1}{1 + \frac{p_T}{1-p_B} \frac{S}{C(T_I)}} \end{aligned} \tag{37}$$

Set associativity is modeled in a same manner as a frame cache. The miss ratio is obtained as follows.

$$\begin{aligned} m_I = \frac{C_B(T_I)}{T_I} & \left(\frac{1}{1 + \frac{p_B}{1-p_B} \frac{C_B(T_I)}{C(B)} \frac{S}{C(T_I)}} \right)^A \\ & + \frac{C_T(T_I)}{T_I} \left(\frac{1}{1 + \frac{p_T}{1-p_B} \frac{S}{C(T_I)}} \right)^A \end{aligned} \tag{38}$$

Informations required to model an instruction cache are, the average thread length T_I, average code block size B, the average number of threads per code block N_T, the code block working set function $W_B(t)$ and a thread working set function $W_T(t)$. The only dynamic information required is just these two working set functions.

6 Conclusion

Whether fine-grain multithreading becomes a truly useful compiling discipline for distributed memory parallel processors depends upon many factors. In this paper, we have at least shown how the technique influences local memory reference behavior in two different parallelism regimes — loop-based parallelism and recursive tree-like parallelism. Surprisingly, we discovered that the reference streams could be characterized by a working set function which is similar to those associated with uniprocessor, single threaded programs. By using this working set function as a basis for an analytic model of cache behavior, we were able to derive a set of useful relations that accurately predict frame and instruction cache miss ratios.

It seems to us that the most important areas of future research lie in understanding the tradeoffs in scheduling ready-to-run data threads. While we have explored very simple LIFO and FIFO strategies, there is a very rich space of possible structures. Our belief is that perhaps the best scheduling algorithms could be generated by the compiler as part of the code generation process itself. Overall, we are encouraged that a fine-grain multithreading compiling discipline is compatible with the local memory organizations of future generations of high performance microprocessors.

References

[1] Wolf-Dietrich Weber and Anoop Gupta. Exploring the Benefits of Multiple Hardware Contexts in a Multiprocessor Architecture: Preliminary Results. In Proceedings of 16th Annual International Symposium on Computer Architecture, IEEE, June 1989, pages 273-280.

[2] Anant Agarwal. Performance Tradeoffs in Multithreaded Processors. IEEE Transactions on Parallel and Distributed Systems, Vol. 3, No. 5, September 1992, Pages 525-539.

[3] David E. Culler, Anurag Sah, Klaus Erik Shauser, Thorsten von Eicken and Jhen Wawrzynek. Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine. In Proceedings of 19th Annual International Symposium on Computer Architecture, IEEE, June 1991, pages 164-175.

[4] Charles E. Leiserson, Z. Abuhamedeh, D. Douglas, C. Feynman, M. Ganmukhi, J. Hill, W. Hillis, B. Kuszmaul, M. Pierre, D. Wells, M. Wong, S. Yang and R. Zak. The Network Architecture of the Connection Machine CM-5. In Proceedings of ACM Symposium on Parallel Algorithms and Architectures, 1992.

[5] R. Alvenson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield and B. Smith. The Tera Computer System. In Proceedings of International Conference on Supercomputing, June 1990.

[6] William J. Dally, L. Chao, A. Chien, S. Hassoun, W. Horwat, J. Kaplan, P. Song, B. Totty and S. Wills. Architecture of a Message-Driven Processor. In Proceedings of 14th Annual International Symposium on Computer Architecture, IEEE, June 1987, pages 189-196.

[7] Anant Agarwal, Beng-Hong Lim, David Kranz and John Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In Proceedings of 17th Annual International Symposium on Computer Architecture, IEEE, June 1990

[8] Rishiyur S. Nikhil, Gregory M. Papadopoulos and Arvind. *T: A Multithreaded Massively Parallel Architecture. 20th Annual International Symposium on Computer Architecture, IEEE, June 1992, pages 156-167

[9] B. J. Smith. A Pipelined, Shared Resource MIMD Computer. In Proceedings of 1978 International Conference on Parallel Processing, 1978, Pages 6-8.

[10] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, Klaus Erik Shauser. Active Messages: a Mechanism for Integrating Communication and Computation. In proceedings of 20th Annual International Symposium on Computer Architecture, IEEE, June 1992, pages 256-266.

[11] Bob Boothe, Abhiram Ranade. Improved Multithreading Techniques for Hiding Communication Latency in Multiprocessors. In proceedings of 20th Annual International Symposium on Computer Architecture, IEEE, June 1992, pages 214-223.

[12] Stephen W. Keckler, William J. Dally. Processor Coupling: Integrating compile Time and Runtime Scheduling for Parallelism. In proceedings of 20th Annual International Symposium on Computer Architecture, IEEE, June 1992, pages 202-213.

[13] Monica S. Lam, Edward E. Rothberg and Michael E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In Proceedings of 19th Annual International Symposium on Computer Architecture, IEEE, June 1991, pages. 63-74.

[14] Rafael H. Saavedra-Barrera, David E. Culler and Thorstem von Eicken. Analysis of multithreaded architectures for parallel computing. In proceeding of 2nd Annual ACM Symposium on Parallel Algorithms and Architectures, IEEE, July, 1990, pages.169-177.

[15] Peter J. Denning. The Working Set Model for Program Behavior. Communications of the ACM, Vol. 11, No. 5, November 1968, Pages 323-333.

[16] Dominique Thiebaut. On the Fractal Dimension of Computer Programs and its Application to the Prediction of the Cache Miss Ratio. IEEE Transactions on Computers, Vol. 38, No.7, July 1989, Pages 1012-1026.

[17] Makoto Kobayashi, Myron MacDougall. The Stack Growth Function: Cache Line Reference Models. IEEE Transactions of computers, Vol. 38, No. 6, June 1989, pages 798-805.

[18] Anant Agarwal, Mark Horowitz and John Hennessy. An Analytical Cache Model. ACM Transactions on Computer Systems, Vol. 7, No. 2, May 1989, Pages 184-215.

[19] Harold S. Stone. High-Performance Computer Architecture. 2nd-edition. Addison-Wesley, 1990, page 81.