An Analysis of Memory Reference Behavior in A Tightly Coupled Multithreaded Architecture

Masato Motomura*
Gregory M.Papadopoulos†

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

December 9, 1993

Abstract

Multithreaded(MT) architecture has been drawing attention because of its capability of hiding long memory latency by switching among threads. In a tightly coupled MT architecture, each thread comes from one of code blocks in a single program. Tightly coupled MT is quite attractive because it can fully utilize fine grained parallelism that exists in a single program. However, possible impact of this architecture on programs' memory reference locality has not been studied well. In this paper, we present the results of trace simulation study on the locality shown in a tightly coupled MT architecture. Results shows LRU stack locality model found in single thread arhitectures also holds for the architecure. We discuss the relation between parallelism and locality based on this model. The influence of thread scheduling is also discussed. Furthermore, we build an analitical cache model based on the analysis of the locality behavior. The model quantitatively predicts simulated cache miss ratio quite well. We also propose as adequate cache organization and thread scheduling strategy to obtain a better hit ratio in a tightly coupled MT architecture.

*On leave from System ULSI Research Laboratory, Microelectronics Research Laboratories, NEC Corporation. Phone: 81-427-71-xxxx, Fax: 81-427-71-xxxx, 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

1.1 Paper Overview

In this paper, we will closely investigate this issue by utilizing both trace driven simulation and analytical method. In section 2 we present a tightly coupled MT architecture and its execution model which lays a basis of our study. Section 3 describes trace driven simulation set up. Results of the simulation that shows relation between parallelism and locality are presented in Section 4 with an in depth analysis. Section 5 gives an analitical cache model in a tightly coupled MT architecture along with simulated cache miss ratios. The model predicts simulation results quantitatively very well. Section 6 gives a thought for an adequate cache organization and a thread execution scheduling in a tightly coupled MT architecture.

2 Tightly coupled MT execution model

2.1 *T Execution Model

3 Trace Driven Simulation

Previous simulation studies of MT architecture have been always using traces obtained by artificially interleaving reference streams from several contexts [1][3] simply because real trace was not available . Though this kind of approach has its meaning for loosely coupled MT architectures, it is impossible to capture memory reference behavior of a titghtly coupled MT model in this way. As explained in Section 1, context switch in the architecuture is deeply involved in its fine grained parallel execution model: it does not make sense to treat number of contexts or context switch intervals as ad hoc parameters. Thus it is really important to get real multithreaded memory reference traces for the study of tightly coupled MT architectures. Our study, as far as the authors know, is the first try to explore general behavior of memory references in this architecture by utilizing real traces.

High level *T simulator has been developed at Motorola Cambridge Research Center [?] for the purpose of verifing several aspects of *T architecture in an early stage of its design. We used this simulator for getting memory traces, and ran our own simulator on them to collect statistics. We first explain the *T simulator briefly. This simulator uses Id language and TAM compiler [5] as its substrate. A program written in Id is compiled by TAM compiler into a threaded code. TL0, a high level machine language for TAM, is used for the genterated code. The simulator simulates dynamic scheduling of this code in multiprocessor environment by applying a *T execution model explained in Section 2.1. Memory reference traces for each processor is obtained by substituting RISC type instruction for each of scheduled TL0 instructions. This simulator is farely high level and is not designed for predicting absolute *T performance. Since the purpose of our study is to explore general features found in a tightly coupled MT architecture, this has little impact for the quality of results.

3.1 Charactersitics of Traces

We ran following two benchmark programs for our study.

Gamteb: An Monte carlo simulation program of photons traversing through a carbon rod. It has one outermost loop, and the code inside consists of banch of code blocks, whose calling path depends on a number randomly generated for each iteration of the loop.

Matrix Multiply(MMP): A triply nested loop program calculating multiplication of two matrixes. No blocking is used in this program.

We ran above programs on various system configurations, data size, and loop bounds. We report here simulation results on 8 node configuration, 20,000 photons for Gamteb and 600 $\times$ 600 matrix for MMP. Three kinds of loop bounds are experimented for each program: 50, 200 and 800 for Gamteb, and 25, 50 and 100 for MMP. Note that loop bounds in MMP is applied for all of the three loops. Furthermore, to see how scheduling infuluence a program's memory reference behavior, we conducted experiments on thread scheduling policies: using two kind of scheduling policies, LIFO and FIFO, for local continuations. A scheduling of global continuations is fixed to FIFO in order to maintain FIFO ordering of a network. The two programs have opposite characteristics in a sense that Gamteb has only one loop, and MMP is heavily looped. It is quite interesting to see how they behave differently in accordance with variations of a loop bound and/or a local continuation scheduling policy. In total, the number of run we report here is 12.

(NOTE: Decralaiton of doing simulation by changing loop bound and scheduling policy in Introduction is assumed)

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 data cache, however, this program is a good test of locality behavior, because it is well studied that to get high data cache hit ratio for the program is quite tricky [4]. In both programs, the average length of syncronization threads, around 5 instructions, are considerably shorter than those of Data threads, which are well above 10 instructions. This is an expected result from a nature of each type of threads. The average number of syncronization threads per code block, however, is larger than one may imagine. A compiler could be improved to decrease this number by compiling away unnecessary syncronization event.

Table 2 shows trace length for two runs out of 12 runs that we explore in this paper. Other runs for each benchmark has essencialy the same size. The traces are not the ones

Benchmark Name		Gamteb	MMP
Program size (Instruction)		24,921	4,238
Number of Code Blocks		51	13
Data Threads	Avg. Number (/CB)	16.76	14.69
	Avg. Length (Instruction)	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 count.

Benchmark Name	Scheduling Policy	Loop Bound	Number of Threads	Instruction References	Frame References	I_structure References
Gamteb	LIFO	200	0.69M	7.95M	3.66M	0.18M
MMP	LIFO	50	0.81M	9.40M	4.60M	0.34M

Table 2: Trace lengh examples.

for all nodes from the begining of the execution to the end of execution, mainly because the lack of time and disk space to accomodate full traces. Instead, we recoreded memory reference trace for one of 8 nodes until the statistics we report later has reached stable state and the traces are long enough for simulations of caches smaller than 64KB. For the purpose of justifing using traces for one node, we verified that statistics variations between nodes are negligible. We will discuss other aspects of possible drawbacks caused by this trace collection method wherever necessary. About the length of traces, all refereces except I_structure references are long enough to simulate desired size of caches. Since I_structure references are not a primary concern for our local cache study, though we will report cache miss rate result later, this lack of sufficient trace lengh is not a problem. I_structure references experience long network latency, which makes caching behavior less important.

Figure 1 shows processor utilization of each run by dividing processor time into Data thread execution time, Syncronization thread execution time and Idle time. This graph takes into account only network latency for calculating Idle time. The large fraction of idle time in MMP can be basically explained by the fact that only a small amout of calculation is done per one I-structure reference in this program. There isn't significant difference in processor utilization between LIFO and FIFO scheduling, though slight improvement can be found for Gamteb. Caution should be used in dictating Figure 1 not only because high level nature of our simulation environment but also because we are not using full traces. As is well studied, amount of parallelism in a program, which strongly affects idle time thorough the latency torelance mechanism, changes a lot along phases of program execution [?]. Thus

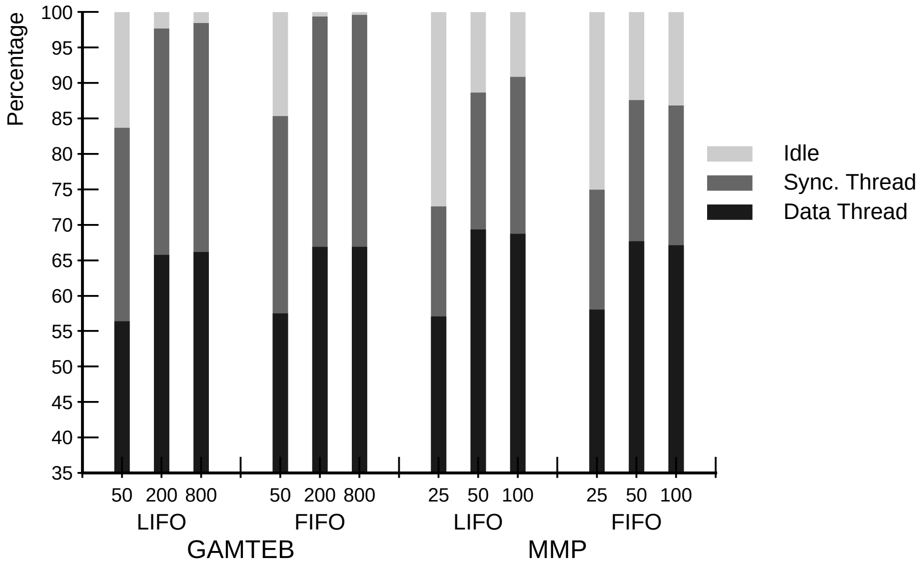


Figure 1: Processor utilization for 12 runs. The processor time is devided into Idle time, Syncronization thread execution time and Data thread execution time. Note that y-axis minimum is not 0.

Figure 1 should not be thought as predicting performance of these programs. 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 [6].

4 Local Memory Reference Behavior Analysis

Since we are seeing memory references no one has seen before, it is quite important to closely investigate into them in order to understand cache memory behavior. In this section, frame and instruction memory references are analyzed separately. We will show how notion of working sets, which is a key for understanding cache behavior in our tightly coupled architecture, can be defined for each of two references.

4.1 Frame Memory Reference

Before seeing actual statistics we have got from the trace simulation, it may be beneficial to overview expected relation between frame statistics and scheduling policy or loop bounds quickly. Generally speaking, LIFO policy schedules an execution of threads by depth first order in a frame activation tree explained in Section 2.1, whereas FIFO policy schedules it by width first order. It means higher locality of frame references should be found in LIFO scheduling than FIFO scheduling. Larger loop bounds increases parallelism, thus may distract locality.

Table 3 shows frame statistics for each run. Frame size is about 80 8-Byte words for Gamteb and 20 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 life time, time from an allocation to a deallocation of a frame measured as a total number of executed threads in a processor, strongly depends on a scheduling policy and a loop bound. Since smaller life time means computation focuses on smaller set of frames rather than keep allocating new frames, this correlation is an expected result. There is a significant difference between Gamteb and MMP in frame execution time to life time ratio. Here frame execution time is a number of thread executed relative to that frame during its life time. Comparatively, MMP is working on smaller size of frame for much longer time than Gamteb, which implicates better frame cache hit ratio for MMP. A number of extant frames, i.e., an average number of allocated frames at a time, strongly depends on loop bounds as frame life time does. However, it does not show any significant difference among two schedulings. The rightmost column shows context switch interval according to a definition given in Section 2.1. Apparently LIFO scheduling gives longer interval, which is again an expected result. One thing to note here is that the fact context switch interval in Gamteb is longer than MMP. It is surprising given that other statistics shows better locality for MMP. This means simple view of treating context switch interval as a measure of locality [1][3] does not work in our MT model. We should find another metric of locality.

What kind of locality can we expect for frame references? This is the fundamental question we want to answer here. As is well known, there are two kinds of locality generally found in any memory references: spatial locality and temporal locality. A frame can be seen as a mechnism to maintain spatial locality of data references in parallel executions of a program, since it is a small set of local storage that is used during an invocation of a code block. It means both spatial and temporal locality will be maintained well, if frame memory references are focused around a few set of frames. In other words, the queston posted above is reduced to a question about temporal locality among frame pointers (FPs) in dynamically scheduled continuations.

One interesting way of investigating this temporal locality is to see correlation between a frame being referenced and a frame referenced in recent past. For this sake, we measured frame LRU stack jump probabilities. Here, a frame LRU (Least Recently Used) stack is defined as a stack which records fixed amount of FPs from most recently used ones. The higher the jump probalilities for recently used frames are, the more temporal locality a program will experience. This way of investigating locality in frame references is analogas to a tequeneque used for studying paging behavior [7][9]. Figure 2 shows results of the measurement for the LRU stack of 50 deep. The result for both programs are quite telling about nature of their locality. For Gamteb, especially in LIFO scheduling, power law behavior between jump probability and stack depth is observed. The power constant is about -2.1. Surprisingly, at least at a first glance, this constatnt is almost indepentent on

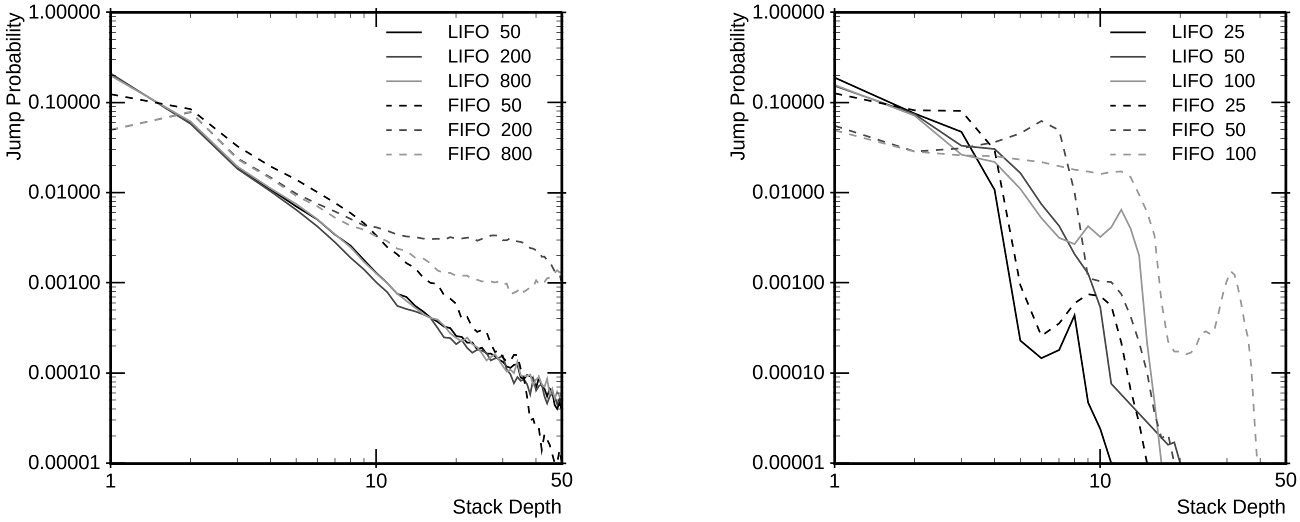
Benchmark Name	Scheduling Policy	Loop Bound	Frame Size	Frame Life Time	Frame Exec. Time	Num. of Extant Frames	Context switch Interval
Gamteb	LIFO	50	80.25	2975.81	62.92	63.38	2.70
		200	80.86	5294.50	62.94	161.49	2.86
		800	81.61	6559.92	64.06	372.96	2.65
	FIFO	50	81.21	1846.34	63.91	39.51	1.86
		200	79.06	8044.61	62.25	169.95	1.40
		800	80.47	15445.56	62.27	374.81	1.41
MMP	LIFO	25	17.45	11306.96	2671.69	18.21	2.24
		50	19.14	20018.77	2469.50	43.01	2.00
		100	20.30	32025.45	2339.11	92.25	2.10
	FIFO	25	17.24	12311.56	2512.49	21.86	1.48
		50	18.45	22778.21	2461.87	43.12	1.14
		100	19.53	46450.58	2368.63	89.93	1.13

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

loop bounds. Too large loop bound allocates frames more than necessary, as apparent from Table 3, however, a locality of frame reference itself is not affected in this case. This can be understood as a consequence of depth first execution of LIFO scheduling, though it is still a little surprising that fixed FIFO scheduling for global continuation seems to have a little impact on this characteristics. For FIFO scheduling, the power law applies less acurately and a behavior for each loop bound shows difference as they deviate from the power law. The loop bound dependency is quite natural. MMP shows more complicated curves: simple structure of a program manifest itself in abnormalies in the jump probabilities. However, these curves are generally characterized by initial rather constant probalities, a sharp drop after that and a bump following them. The constant portion extends to deeper region as a loop bound is increasedand scheduling is changed from LIFO to FIFO.

Figure 3 provides a more detailed view of a same probability for one run from each program's runs. In this Figure, the probability is devided into four categories by a combination of two conditions: from which type of thread and to which type of thread. It can be seen that Data thread shows better locality than syncronization thread in both programs: jump probability to a data thread is higher than that to a syncronization thread in recently used frames, regardless of which type of thread is being executed. This reflects stocastic nature of syncronization thread execution. Syncronization thread is executed mostly in response to a message arrived from network.

Figure 2 and 3 shows that an execution of threads tends to have some set of frames on which threads are intensively scheduled. This tells us a working set model, which is



(a) Gamteb (b) MMP

Figure 2: Frame LRU stack jump probability for (a) Gamteb and (b) MMP.

well etsablished as a memory reference model for conventional architectures [8], holds also for frame memory reference behavior in our parallel architecture. Here 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 4 shows measured $W_F(t)$ for all of runs. LIFO scheduling for Gamteb shows again power law behavior, which does not depend on loop bounds neither. MMP's frame working set function shows saturation after initial rise whose slope is compatible with Gamteb. Intuitively, the power law for Gamteb implicates that this program's working set has "transient" nature: some portion of a working set is continuously changing along with an execution of a program, which results in a continuously growing frame working set function. On the otherhand, saturation behavior for MMP means that this program's working set has "recurrent" nature: the working set is rather closed in a sense that same fixed number of frames are used for some interval of time.

It is easy to find direct correlation between frame LRU stack jump probabilities and frame working set functions. If you rotate a Figure 4 for 90 degrees, you will find similar curves to Figure 2 with smoother charactersitics. For instance, saturation of MMP's working set size around 10 frames can be understood as a direct consequence of the fact that jump probabilities for this program drops around stack depth 10. Actually, working set function can be derived from LRU stack jump probabilities by simple calculations as shown in [9]. Let J_i be a jump probability to depth i of a frame LRU stack. The required time for a working set to grow size of n, which corresponds to an inverse function of a $W_F(t)$, is calculated as,

$$\sum_{j=1}^{n} \frac{1}{1 - \sum_{i=1}^{j} J_i}.$$

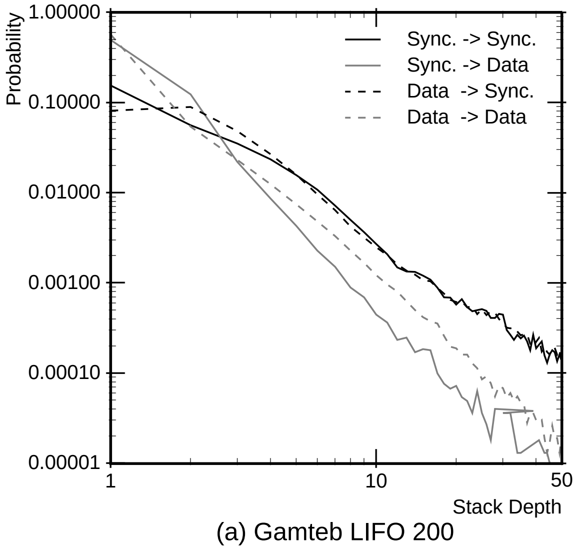


Figure 3: Frame LRU stack jump probability for (a) Gamteb LIFO 200 and (b) MMP LIFO 50. Jump probability is categorized into 4 curves, depending on data thread or syncronization thread.

Calculation of this equation fit measured curves farely well.

The range of power constant for Gamteb, from 0.42 to 0.55 overlaps the values found in conventional architectures, which ranges from 0.484 to 0.544 [10]. This implicates locality level in frame memory reference is about the same as memory references in conventional architectures, though data points are not enough to judge. One thing to note is that we are seeing only frame references which corresponds to stack references in conventional architectures whereas above numbers for conventional architectures incude all memory references.

4.2 Instruction Memory Reference

For instruction memory references, scheduling influence should have diffent aspect from the infuluence for frame memory references. Intuitively, there was no chance that FIFO policy works better for frame reference because of its width first scheduling nature. However, at least there seems to be a chance that FIFO works better for maintaining locality for instruction references, because in our parallel execution model, same code block might be invocated in parallel, which might assist instruction reference locality in FIFO scheduling.

What is a good way to see locality of reference for instructions? Unfortunately, there is no single framework as a frame in a former case. A code block gives us a good measure of spatical locality, but not for temporal locality. The reason for this difference is quite simple: threads that belong to a same code block and a same frame never share a code

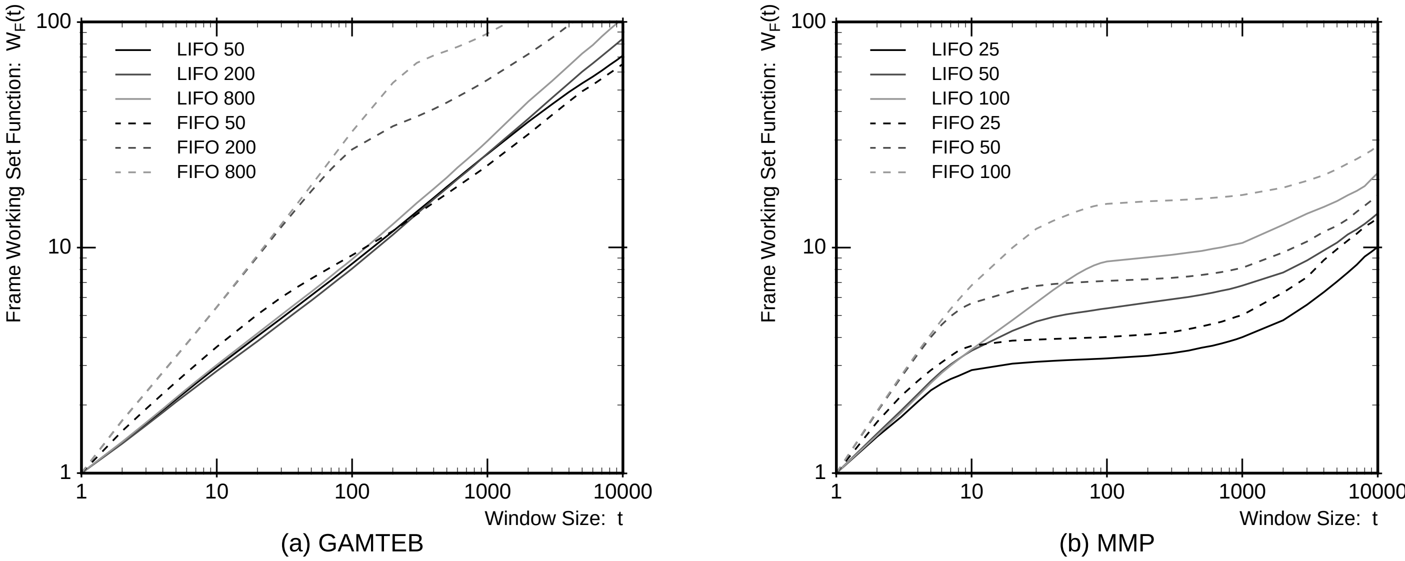


Figure 4: Measured frame working set function $W_F(t)$ for (a) Gamteb and (b) MMP. This function shows how many new frames are scheduled for execution within a given period of time window. Time t is measured by dynamically executed number of threads.

itself though they share a frame as a common resource. Thus in order to see locality behavior in instruction references, we have to separately investigate how code block and threads are scheduled. We will report measurement of two functions, *code block working set function* $W_C(t)$ 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. Though we measured corresponding LRU stack jump probabilities, we omit the results here because their direct relationship with working set function has already been clarified.

Figure 5 shows measured $W_C(t)$. Since number of code blocks is small and limited number, both of the curves show saturation. Gamteb shows again power law behavior with lower power constant ranges from 4.0? to 5.0?. In MMP, working set is limited to almost 1 code block, which may be a code block inside of inner most loop, for a long period. From this figure, it can be seen that FIFO shows poor spatial locality in both cases. This can be understood in following way. Threads which have same frame pointer are guarandeed to belong same code block. So locality of frame reference is almost directly dictated for locality in code block references, though as explained above, this only gurantees a spatial locality.

Figure 6 shows measured $W_t(t)$ which shows how temporal locality in instruction references is exibited for each program. It is easy to find out poor temporal locality for Gamteb. The power constant here, 0.992, is almost 1.0, which means nearly zero thread is re-executed during this period, regardless of scheduling and loop bounds. This is a disappointing but still 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 be working

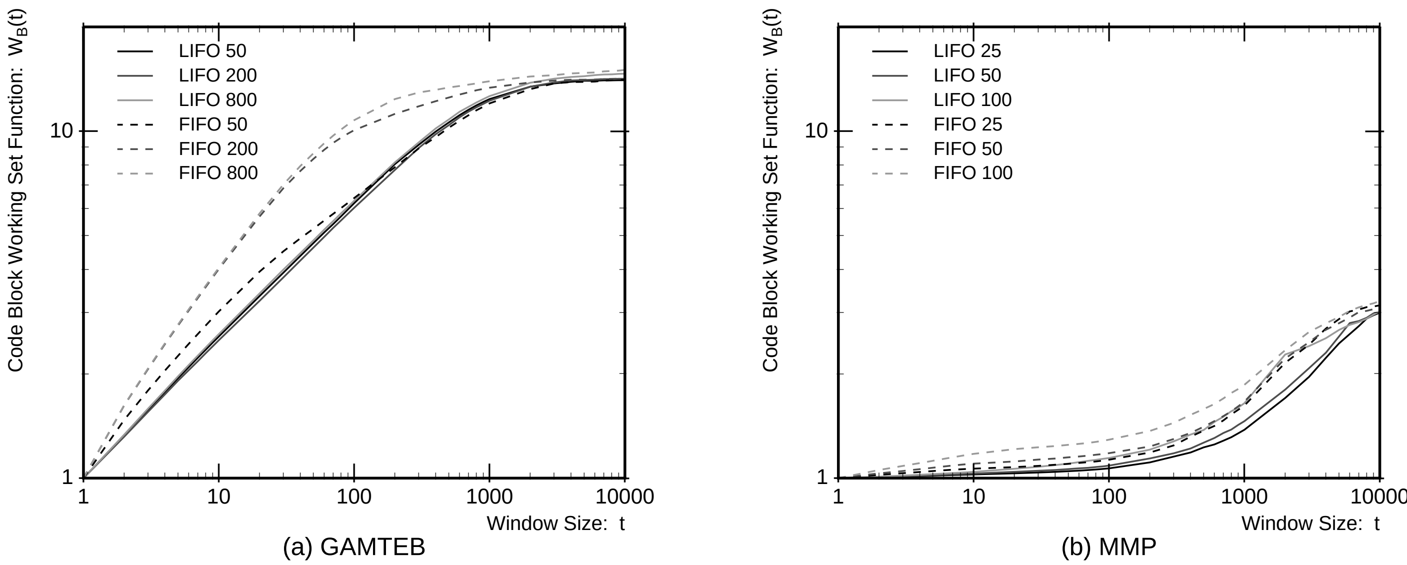


Figure 5: Measured code block working set function $W_C(t)$ for (a) Gamteb and (b) MMP. This function shows how many new code blocks are scheduled for execution within a given period of time window.

better for maintaining temporal locality in overall, FIFO performs slightly better in small working set region before reaching a knee around size of 5 threads. This could be attributed to an argument in the begging of this section. Though it is invisible from the figure, actually there is slight improvement in Gamteb too.

We will see how these two locality component affect instruction cache miss ratio in Section 5.4.

5 Tightly coupled Multithread Cache Model

In Section 4, we have shown working set function can capture locality behavior of both frame and instruction reference quite well. In this section, we will buid up analytical cache model for a tightly coupled MT architecture. This model explains measured cache miss ratio quite well, as we will see in a next section. Though our cache model is geared toward explaining cache behavior in a tightly coupled MT architecture, it incorporates new and general idea for using working set function as a basis for calculating cache miss ratio. This idea is readily applicable to any cache modeling whereever working set model applies.

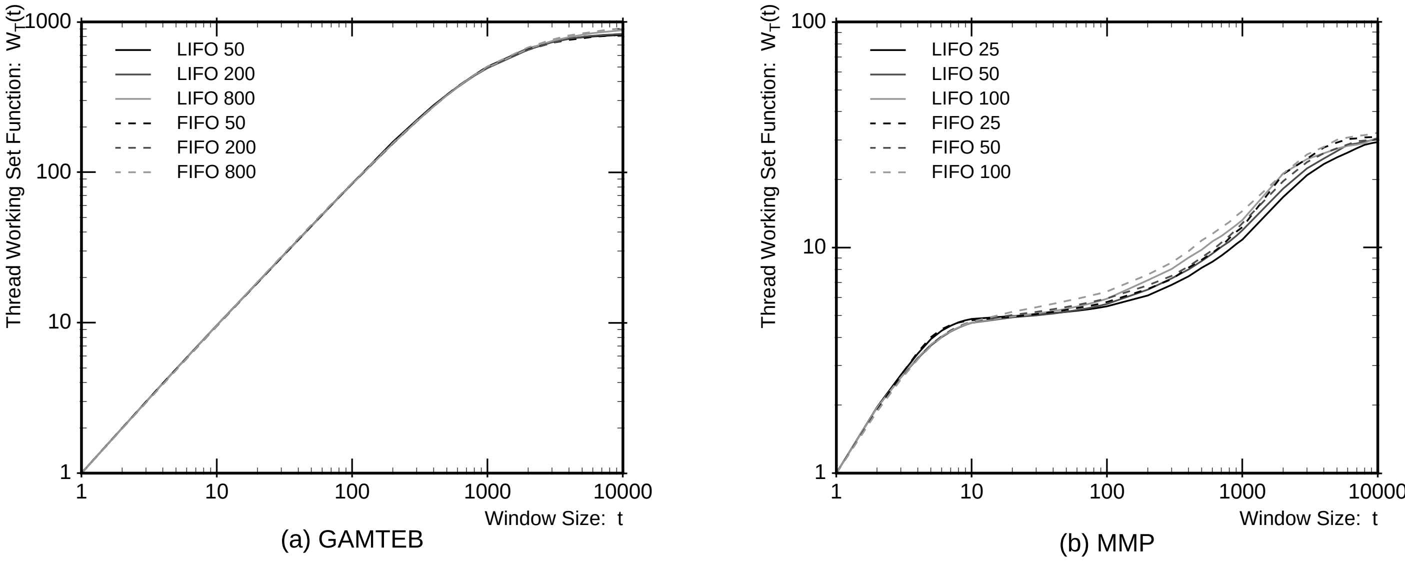


Figure 6: Measured thread working set function $W_t(t)$ for (a) Gamteb and (b) MMP. This function shows how many new threads are scheduled for execution within a given period of time window.

5.1 Previous Work

The most important and most extensive work in modeling cache behavior was done by Agarwal, Hennesy and Horowitz [2]. This work proposed full framework for understanding cache behavior for various cache configurations including changing line size 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 an ad hoc manner. Though they claimed the choice of time granule does not affect model's predection much, we believe there should be some systematic way of deciding this parameter. If a working set is measured by some ambiguous parameter, the meaning of working set itself is also ambiguous. Secondly, one parameter named collision rate, which is measured from traces separately, plays a key roll in their modeling. Since working set model is proven to work well for tightly coupled multithreaded architecture as well as conventional architectures, we want parameters to be derived directly from a working set function. Lastly, LRU algorithm, which has practical importance for set associative cache is not modeled in their work.

Another important work was reported in [10]. This work introduced notion of fractal random walk in the field of cache modeling for the first time. Actually, the power law we observed in LRU stack jump probabilities and in working set functions can be explained by this concept. Our programs show behavior of fractal random walk, just programs in conventional architectures do. The cache model proposed in the paper is superior in a respect that it relates cache miss ratio directly to working set funciton. However, only a model for fully associative cache was proposed in this paper.

In a context of multithreaded architecture, there has been only one work of cache modeling [3]. This work ins a direct descendent of work described in [2]. In this work, multithreading is treated as an extreme case of multiprograming, where context switch interval is not long enough to fully bring its working set into cache. Number of context is treated as an artificiatlly controlable parameter. We already stated that this treatement is not applicable for our target architecture.

5.2 Working Set for Cache Modeling

We start our cache modeling effort with an observation of a notion of working set. This section serves for supplying general idea of our work for readers. Accuracy of terms is sacrificed for this purpose, only in this section.

As we saw in Section 4, and is well known [8], working set function is a continuously growing function of time. This is simply because a program touches new memory localitions as its execution proceeds. This means size of a working set for a program depends on how long it is measured. In other words, in an intuitive notion of working set, i.e., where program is "currently" focusing its computation, a choice of how "currently" changes working set size completely. This is a fundamental difficulty in building perspective cache model based on a notion of working set, which has not been tackled well in previous works.

An important claim we make here is that a time interval for measuring a working set size, τ, should vary depending on cache size. As an intuitive example, suppose we have a memory reference stream which referes, say, 4 consequtive memory locations cyclicaly, i.e, 00, 01, 10, 11, 00, 01, 10, 11, $\cdots$. Let's assume we decide to measure working set at, say, 10-reference interval. Thus working set size of this reference stream is 4, because there are only 4 distinct memory locations that are referenced. If our "cache" can memorize more than 4 locations, it can succesfully accomodate this reference pattern. However, if our "cache" size is less than 4, this cache continuously misses. The cyclic reference pattern does not help at all for the size of cache less than the pattern cycle. It has no meaning whether working set size defined through above ad hoc way is 4 or, say, 10. Put it in another way, different size of cache "sees" references with different "time resolution". A smaller size of cache sees more detailed, shorter period pattern of memory reference streams than a larger size of cache. Decision of τ should incorporate this fundamental view in order to model various size of caches coherently.

We introduce *working set growth rate* p as

$$p = \tau \frac{W'(\tau)}{W(\tau),} \tag{1}$$

where $W(\tau)$ is a working set function at time τ, which gives a working set size by definition of τ, $W'(\tau)$ is a time t derivative of $W(\tau)$. p gives a ratio of newly referenced working set witin $W(\tau)$. That is, $(1-p)W(\tau)$ out of $W(\tau)$ is carried over from a working set in an adjacent τ period. If p is 0, for example, a working set function is not growing at all. If p is

1, there is no overlap in adjacent τ period, which means working set is increasing linearly against time. We can derive a general equation which $W(\tau)$ should satisfy as follows:

$$W(\tau) \times (1 - \ Miss\ Ratio) = r(p) \times Cache\ Capacity, \tag{2}$$

where $r(p)$ is an unknown function we are going to find below. The left hand side of Equation (2) gives size of cache resident portion of a working set, because hit ratio means that amont of portion out of a working set should be cache resident in order to experience cache hit. Thus, Equation (2) simply says a cache resident porion of a current working set occupies $r(p)$ of cache capacity. $(1 - r(p))$ fraction of a cache is left over from references which have drifted out from the current working set. There are two conditions that $r(p)$ should satisify. $r(0)$ should be equal to 1 because working set size is not increasing when $p = 1$. On the other hand, $r(1)$ should be equal to 0 because no memory location is being reused, i.e, left hand side of the equation is 0. $r(p)$ can be chosen arbitrarily provided that it satisfies these two boundaly conditions. Thus we can set simply

$$r(p) = 1 - p. \tag{3}$$

Since miss ratio of a cache is going to be calculated by using $W(\tau)$, Equation (2) 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 boundaly conditions. Mathematicaly, a miss ratio is obtained as a fixed point of self consistent of equations, which we will derive following sections.

This self consistency requirement has been a key to solve the *chiken and egg* type problem of deciding working set size. This technieque is applicable to dicide any type of working set size required in modeling a cache behavior.

Here we can preview why previous study that decides τ in an ad hoc way [2] has made success. Suppose power law applies to a working set function, i.e., $W(t) = \alpha t^{\beta}$. Above defined working set growth ratio is calculated as,

$$p = \tau \frac{\alpha\beta\tau^{\beta-1}}{\alpha\tau^{\beta}} = \beta. \tag{4}$$

Thus this ratio does not depend on τ. It means, as we will see, a miss ratio does not depend on τ neither. This is one example of a sefl-similar behaviors generally observed in fractal random walks. However, a problem is that simple power law does not apply to all cases. In many cases, there are often changes of power constant as time window grows [11]. Moreover, working set function shows saturation sooner or later. In order to obtain cache behavior when working set function shows saturation for that size of cache, cache size dependency of working set size must be introduced to predict cache miss ratio correctly. Note that simulation results that were compared with a model in [2] did not cover this region [11].

5.3 Frame Cache

Now we apply above general idea for modeling 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. This is a quite reasonable assumption for a direct map cache, because it has enough number of sets in cache size ranges of interest. This assumption is quite helpful for simplifying our frame cache model. Secondly, we assume random mapping of FPs on cache lines, and random memory references within a frame. We model only *warm start miss ratio* here, because we are modeling a multithreaded execution model in a single programming environment.

We calculate miss ratio of a frame cache by counting how many references from a thread miss a cache 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

$$m_F(L, A, S) = \frac{M_F(L, A, S)}{T_F}. \tag{5}$$

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 dependency wherever approapriate in order to keep notational simplicity.

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

Generally, each of T_{Fc} references directly gives a miss, which is known as *compulsory miss*, because it is the first access to a line 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 there is no data consistency problem between different frame allocations. Thus, we can define compulsory miss rate 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 straitforward.

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 totl, a frame cache miss ratio is given by

$$m_F = \frac{T_{Fc}}{T_F} m_{Fc} + \frac{T_F - T_{Fc} - T_{Ft}}{T_F} m_{Fi}. \tag{6}$$

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 *cover function* C [2] as

$$C(F) = 1 + \frac{F-1}{L}. \tag{7}$$

Suppose R_F lines of a frame within a working set are resident in a cache on average. Based on an idea in Section 5.2, 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, working set size in this case means a number of frames that constitute a working set. Thus, Equation (2) is dictated into a following equation:

$$W_F(\tau)R_F = (1-p_F)S, \tag{8}$$

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

$$p_F = \tau\frac{W'_F(\tau)}{W_F(\tau)}. \tag{9}$$

Since average fraction of a frame that is not resident in a cache gives an average ratio where references to that frame misses,

$$m_{Fi} = 1 - \frac{R_F}{C(F)} \tag{10}$$

holds. Thus Equation (8) gives a frame working set size as a function of m_{Fi}.

$$W_F(\tau) = \frac{1-p_F}{1-m_{Fi}}\frac{S}{C(F)}. \tag{11}$$

As explained in Section 5.2, Equation (11) 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 and see which frame does this thread belong 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.

$$T_{Fi}(1-p_F)\left(1-\frac{R_F}{C(F)}\right) = T_{Fi}p_F\frac{R_F}{S}. \tag{12}$$

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, increment of R_F in Poisson approximation is given by the left hand side of the equation. Right hand side gives a decrement of R_F, because $\frac{R_F}{S}$ gives a probability that intereference occurs. Thus,

$$R_F = \frac{C(F)}{1 + \frac{p_F}{1-p_F}\frac{C(F)}{S}}. \tag{13}$$

is obtained. m_{Fi} is easily derived from R_F by using Equation (10) as

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

It turns out that p_F and $\frac{S}{C(F)}$ are two important parameters that completely decides this miss ratio. If $p_F = 1$, $m_{Fi} = 1$ as expected. Note, again, that pF is calculated from Equation (9) and Equation (11) through the self consistency mechanism.

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. Steady state equation for R_c is given by

$$C(F)p_r\left(1 - \frac{R_c}{C(F)}\right) = C(F)(1-p_r)\frac{R_c}{S} + C(F)(N_c - 1)\frac{R_c}{S}. \tag{15}$$

Left hand side and a first term of the right hand side can be understood in a same fashion as corresponding terms in Equation (12). These terms are multiplied by $C(F)$ because $C(F)$ distinct lines are referenced within a frame allocation. Second term of right hand side is turbulance 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,

$$N_c = \frac{1}{\tau_L}\sum_{t=1}^{\tau_L} W_F(t). \tag{16}$$

From Equation (15),

$$R_c = \frac{C(F)}{1 + \frac{N_c - p_r}{p_r}\frac{C(F)}{S}} \tag{17}$$

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

$$\begin{aligned} m_{Fc} &= p_r\left(1 - \frac{R_c}{C(F)}\right) + (1-p_r) \\ &= 1 - \frac{p_r}{1 + \frac{N_c - p_r}{p_r}\frac{C(F)}{S}}, \end{aligned} \tag{18}$$

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:

$$m_F = \frac{T_{Fc}}{T_F}\left(1 - \frac{p_r}{1 + \frac{N_c - p_r}{p_r}\frac{C(F)}{S}}\right) + \left(1 - \frac{T_{Fc}}{T_F} - \frac{T_{Ft}}{T_F}\right)\left(\frac{1}{1 + \frac{1-p_F}{p_F}\frac{S}{C(F)}}\right). \tag{19}$$

For this model, we need following average numbers and a function: a number of frame memory references per thread T_F, number of compulsory references T_{Fc}, number of frame references that are covered within a thread T_{Ft}, size of a frame F, 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 staticaly. 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 again by neglecting self-interference on a cache set. Though it was a farely good assumption for direct map cache, this is not necessarily good for set associative cache, because S decreases as A increases for a same size of cache. We use this assumption, however, because it gives us a good insight on cache behavior with a simple mathematical form. Self-interference can be introduced into a model quite easily, but gives more compicated form. Note this assumption gives optimistic miss ratio when inappropriately applied.

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

$$W_F(\tau) = \frac{(1-p_F)}{1-m_{Fi}} \frac{AS}{C(F)}. \tag{20}$$

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

$$T_{Fi}(1-p_F)\left(1-\frac{R_F(1)}{C(F)}\right) = T_{Fi}p_F\frac{R_F(1)}{S}, \tag{21}$$

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

where $1 < i \leq A$. Left hand sides give increment of R_F's, while right hand side gives decrement. Equation (21) is basicaly a same equation as Equation (12). In Equation (22), 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 (21) and (22) are easily solved as,

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

$$R_F(i) = \left(1-\frac{R_F(1)}{C(F)}\right)R_F(i-1). \tag{24}$$

Since R_F is given by a sum of this series, m_{Fi} is calculated by using Equation (10) as

$$m_{Fi}(L, A, S) = m_{Fi}^A(L, 1, \frac{S}{A}), \tag{25}$$

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 $\frac{S}{A}$ lines.

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

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

5.4 Instruction Cache

Instruction cache needs a little different modeling because of two kinds of locality component that should be considered separetely (see Section 4.2). In this section, we will show how two kinds of working set function, 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 miss in order to simplify the notation, because compulsory misses are a negligible component in any practical program for an instruction cache. Since a thread occupies consective 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,

$$m_I = \frac{C(T_I)}{T_I} m_{Ii}, \tag{27}$$

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 devided 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.

$$\begin{aligned} C_T(T_I) &= \frac{1}{N_T} C(B), & (28)\\ C_B(T_I) &= \frac{1}{N_T} \left\{ N_T C(T_I) - C(B) \right\}, & (29) \end{aligned}$$

where N_T is an average number of threads in a code block, and B is an 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, 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, respectivly. Since $C_T(T_I)$ and $C_B(T_I)$ does not overlap each other by their definition, a self consistency equation to give working set size and its measurement time τ in this case becomes as

$$\frac{W_B(\tau)N_T R_B}{1-p_B} + \frac{W_T(\tau)R_T}{1-p_T} = S. \tag{30}$$

Here, p_B and p_T are defined as

$$p_B = \tau\frac{W'_B(\tau)}{W_B(\tau)}, \qquad p_T = \tau\frac{W'_T(\tau)}{W_T(\tau)}. \tag{31}$$

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

$$(1-p_B)\frac{C_B(T_I)}{C(B)}(C_B(T_I)-R_B) = p_B C(T_I)\frac{R_B}{S}, \tag{32}$$

$$(1-p_T)(C_T(T_I)-R_T) = p_B C(T_I)\frac{R_T}{S}. \tag{33}$$

In a left side of Equation (32), $\frac{C_B(T_I)}{C(B)}$ gives a probability that a thread references $C_B(T_I)$ portion of a code block. If it does, cache resident portion increases from R_B to $C_B(T_I)$ because of consequtave nature of instruction references. 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 working set may cause interference. In Equation (33), 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

$$R_B = \frac{C_B(T_I)}{1+\frac{1-p_B}{p_B}\frac{C(B)}{C_B(T_I)}\frac{C(T_I)}{S}}, \tag{34}$$

$$R_T = \frac{C_T(T_I)}{1+\frac{1-p_B}{p_T}\frac{C(T_I)}{S}}. \tag{35}$$

Thus an instruction cache miss ratio m_I is given as

$$\begin{aligned} m_I &= \frac{C(T_I)}{T_I}\left(1-\frac{R_B+R_I}{C(T_I)}\right) \\ &= \frac{C_B(T_I)}{T_I}\frac{1}{1+\frac{p_B}{1-p_B}\frac{C_B(T_I)}{C(B)}\frac{S}{C(T_I)}} + \frac{C_T(T_I)}{T_I}\frac{1}{1+\frac{p_T}{1-p_B}\frac{S}{C(T_I)}} \end{aligned} \tag{36}$$

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

$$m_I = \frac{C_B(T_I)}{T_I}\left(\frac{1}{1+\frac{p_B}{1-p_B}\frac{C_B(T_I)}{C(B)}\frac{S}{C(T_I)}}\right)^A + \frac{C_T(T_I)}{T_I}\left(\frac{1}{1+\frac{p_T}{1-p_B}\frac{S}{C(T_I)}}\right)^A \tag{37}$$

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

6 Application of Cache Model

We will show how our model reproduce simulatd cache miss ratio. Then we simplify our model to predict range of cache miss ratio for a tightly coupled MT architecture.

6.1 Comparison with Simulated Results

6.2 Cache Performance Prediction

Since we understand detailed cache behavior of multithreaded architecture, we are free to reduce complexity of the model and see what are the expected performances for caches in multithreaded architecture, in general. Simplification of the model is as follows:

- Working set function obeys simple power low. We assume

$$W_F(t) = \alpha_F t_F^{\beta},$$

$$W_B(t) = \alpha_B t_B^{\beta},$$

$$W_T(t) = \alpha_T t_T^{\beta},$$

- Compulsory misses for a frame cache is treated as a constant.

As already explained in Section 5.2, miss ratio does not depend on τ and $W(\tau)$ when simple power low applies.

Figure **??** shows predictions of a simplified model for a variety of choices of parameters.

What parameters affects cache hit ratio. scheduling, etc

References

[1] Wolf-Dietrich Weber and Anoop Gupta. Exploaring 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, pp. 273-280.

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

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

[4] 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, pp. 63-74.

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

[6] 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, pp.169-177.

[7] Rollins Turner and Bill Strecker. Use of the LRU Stack Depth Distribution for Simulation of Paging Behavior. Communications of the ACM, Vol. 20, No. 11, November 1977, Pages 795-798.

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

[9] Makoto Kobayashi and Myron H. MacDougall. The Stack Grouth Function: Cache Line Reference Models. IEEE Transactions on Computers, Vol. 38, No.6, June 1989, Pages 798-805.

[10] 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.

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