Effective Dispatching for Simultaneous Multi-Threading (SMT) Processors by Capping Per-Thread Resource Utilization

Tilak K. D. Nagaraju, Caleb Douglas, Wei-Ming Lin, Eugene John

Abstract

Simultaneous multithreading (SMT) provides a technique to improve resource utilization by sharing key datapath components among multiple independent threads. When critical resources are shared by multiple threads, effective use of these resources proves to be the most important factor in fully exploiting the system potential. Allowing any of the threads to overwhelm these shared resources not only leads to unfair thread processing but may also result in severely degraded overall performance. How to prevent idling threads from clogging the critical resources in the pipeline becomes a must in sustaining system performance. In this paper, we show that, by simply setting a cap on the number of the critical Issue Queue (IQ) entries each thread is allowed to occupy, the system performance is easily enhanced by a significant margin. An even more pronounced advantage of the proposed technique over other advanced dispatching algorithms is that the performance gain is obtained with very minimal additional hardware required.

Index Terms—Computer Architecture, Superscalar, Simultaneous Multi-threading

I. Introduction

Based on the traditional superscalar processors, Simultaneous Multi-Threading (SMT) offers an improved mechanism to enhance overall system performance without having to invest an proportional amount of extra hardware. In a conventional superscalar processor, not only the functional units are not close to be fully utilized with only one thread running at any time, switching from one thread (task) to another involves the intervention of the operating system which leads to extra overhead in CPU power. Simple multithreaded processors remove the necessity of task switching by OS but still does not fully exploit the capacity of the functional units since a single thread does not usually have enough instruction-level parallelism. A coarse-grained multithreaded processor does not interleave instructions from different threads in processing, while a fine-grained one allows for a cycle-to-cycle interleaving from different threads’ instructions, but neither permits instructions from different threads to be issued in the same clock cycle. SMT takes this one step further in order to exploit the full potential of the functional units. The most common characteristic of SMT processors is the sharing of key datapath components among multiple independent threads, which ensures improved resource utilization. SMT also exploits not only thread-level parallelism (TLP) among the various threads [10], [11], but also equally concentrates on the advantages available at the instruction-level parallelism (ILP) in each thread. Subsequently, due to the sharing of resources, the amount of hardware required in an SMT system is significantly less than employing multiple superscalar machines while achieving similar performance. Most common resources shared in an SMT system include components that are control-complexity-wise easier to share (such as cache memory and physical register bank), and those that are cost-wise better to share (such as Issue Queue (IQ), reservation stations and various functional units). On the other hand, other more thread-specific component (such as Re-Order Buffer (ROB)) along the datapath are assumed to remain per-thread ownership. Due to the requirement in resource sharing, these hardware components tend to remain busy in order to accommodate more instructions from all threads. Although allowing these instructions to share these resources ensures the full performance potential afforded by SMT [6], it tends to induce extra control complexities in managing the critical timing path and the processors cycle. To retain the
exploitation of both TLP and ILP, a necessary solution must be introduced in order to minimize the complexity among these shared resources without affecting the ILP exploitation significantly. At the same time, proper intelligence has to be incorporated into the resource sharing mechanism to ensure that threads share these components in the most efficient and fair manner.

II. Instruction Dispatch

There have been many different terminologies adopted for instruction pipelining stages (e.g. issue, dispatch, etc.) in a superscalar system, and their references became even more ambiguous in an SMT system in which more resource sharing is required than in a superscalar system. For example, instruction “issuing” has been referred to different stages of processing by different articles. Throughout this paper, we choose to adopt the terminologies used by most SMT articles.

In a typical single-thread superscalar system, instructions are “dispatched” from ROB into the reservation stations (either centralized or functional unit-specific) when space is available and then “issued” to the corresponding functional unit whenever the issuing conditions are met, i.e. operands are ready and the requested functional unit becomes available. However, in a basic SMT system, each thread has its own ROB and instructions from these threads-specific ROBs have to “compete” for a shared Issue Queue (IQ) through a dispatching scheduling algorithm. This IQ can be considered as Centralized Reservation Station not only shared among the functional units but also shared among the threads in real time. A basic functional block diagram of this basic design is depicted in Figure 1.

Due to the significantly large size of each IQ entry, the number of entries in this shared resource usually is much smaller than the number of ROB entries. Having its output sent into a tightly shared resource, the instruction dispatch stage is considered one of the most critical stages that dearly affect the overall system performance. There have been many different techniques in scheduling instructions to be dispatched into IQ. On the other hand, instructions from higher ILP (or simply “faster” in instruction issuing) should be given a higher priority in utilizing the IQ. There have not been many research results on how to share the IQ in a more time-adaptive manner allowing different threads to utilize (occupy) the IQ in an “on-demand” basis. In [5], an adaptive technique is proposed to allow all ROBs to be “shared” among threads in order to accommodate threads that are more active with extra ROB entries borrowed from the ROBs from less active threads, in which “activeness” of a thread is based on a ratio between number of issue-bound and commit-bound instructions in a thread’s ROB. Control complexities involved in sharing the ROBs may be the most inhibitive factor in justifying the performance gain from such an approach. Some other more advanced scheduling techniques, such as the one presented in [12], combine more information from different stages in the pipeline to optimize the scheduling/dispatching result, albeit requiring significantly more hardware and control logic.

In this paper, we choose to retain the per-thread ROBs without any sharing among them, and rely on a simple scheduling algorithm in dispatching instructions from different threads. Our analysis shows that activeness of a
thread can be fluctuating very unexpectedly in time, due to several factors, e.g. cache misses, branch miss-predictions, write port latencies, etc. A thread that has been active can suddenly becomes “inactive” (or, more precisely, much less active) due to any of these reasons and stays “idle” in the pipeline for a long duration of time. A write cache miss can easily delay the in-order commit stage while a read miss can significantly slow down the pipeline due to data dependencies. A branch miss-prediction requires all speculative instructions to be flushed out and subsequently the activeness of the thread suffers before the correct path of instructions is re-established. To make it worse, these threads that have been just recently more active (than other threads) tend to occupy more shared resources (for example, the IQ, reservation stations, among others) than others. When these threads suddenly become inactive, the shared resources typically cannot be released for other threads to use, mainly due to various system or operating limitations inhibiting it from doing so. Consequently, there could be only a very limited amount of shared resources for other threads to share, and thus limiting the potential performance. While there could be many different intelligent ways to redistribute the resources, they tend to be either very costly in hardware or simply infeasible in implementation for real-time operation due to the excessive logic required to realize the intelligence. To be able to further improve the system resource utilization at this level of high-speed instruction processing, one cannot afford employing any design that involves too much intelligence for real-time implementation.

There have been many non-adaptive dispatching algorithms proposed, including simple round-robin, within-clock-cycle-round robin [2], two-op-block [7], etc. None of these algorithms are assigned with a priority among threads, instead all are based on a pre-assigned thread index order. The basis for performance comparison in this paper will be the traditional simple round-robin dispatching which simply starts dispatching all dispatchable instructions from a thread according the index order and moves to the next thread if the dispatching bandwidth allows for more instructions; the index order is then rotated naturally to the next thread to start in order in the next clock cycle.

III. Simulation Environment

The simulation environment including the simulator and the workloads used for our simulations are first described in this section.

A. Simulator

We used M-Sim [4], a multi-threaded micro architectural simulation environment model, to estimate performance and power analysis of the proposed scheme. M-sim includes accurate models of the pipeline structures such as explicit register renaming, concurrent execution of multiple threads, detailed power estimation using Wattch framework [8], separate Reorder Buffer, and register files which are necessary for an SMT model. The Issue Queue, execution functional units, Load-Store Queue (LSQ) are shared among the threads, but branch predictor is exclusive to each thread. The detailed processor’s configuration is shown in Table I.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machine Width</td>
<td>8 wide fetch/issue/commit</td>
</tr>
<tr>
<td>Window size</td>
<td>16-entry Issue Queue</td>
</tr>
<tr>
<td></td>
<td>48-entry Load/Store queue</td>
</tr>
<tr>
<td></td>
<td>128-entry ROB</td>
</tr>
<tr>
<td>Function Units &amp;</td>
<td>8 Int Add (1/1)</td>
</tr>
<tr>
<td>Latency (total/issue)</td>
<td>4 Int Mult (3/1) / Div (20/19)</td>
</tr>
<tr>
<td></td>
<td>4 Load/Store (2/1), 8 FP Add (2)</td>
</tr>
<tr>
<td></td>
<td>4 FP Mult (4/1) / Div (12/12) / Sqrt (24/24)</td>
</tr>
<tr>
<td>Physical registers</td>
<td>256 integer and floating point</td>
</tr>
<tr>
<td>L1 I-cache</td>
<td>64KB, 2-way set-associative 128 byte line</td>
</tr>
<tr>
<td>L1 D-cache</td>
<td>32 KB, 4-way set-associative 256 byte line</td>
</tr>
<tr>
<td>L2 Cache unified</td>
<td>2 MB, 8-way set-associative 512 byte line</td>
</tr>
<tr>
<td>BTB</td>
<td>2048 entry, 2-way set-associative</td>
</tr>
<tr>
<td>Branch Predictor</td>
<td>2K entry gShare</td>
</tr>
<tr>
<td></td>
<td>10-bit global history per thread</td>
</tr>
<tr>
<td>Pipeline Structure</td>
<td>5-stage front-end (fetch-dispatch) scheduling (for register file access: 2 stages, execution, write back, commit)</td>
</tr>
<tr>
<td>Memory</td>
<td>64 bit wide, 200 cycles access latency</td>
</tr>
</tbody>
</table>

TABLE I. Configuration of the Simulated Processor

B. Workloads

For multi-threaded workloads, we use the mixed SPEC CPU 2000 benchmark suite [1], [3], [8] based on ILP classification. Each of the benchmarks is initialized in accordance with the procedure mentioned in Simpoints tool [9] and then up to 100 million instructions are simulated. Once the first 100 million instructions are committed from any of the threads, simulation is terminated.

In order to categorize multithreaded workloads, each of the benchmarks is simulated individually in a simple scalar environment. As shown in Table II, three types
of ILPs - low ILP (memory bound), medium ILP and high ILP (execution bound) - are identified, and six mixes of multi-threaded workloads are used based on different combinations of ILP types. Mixes 5 and 6 are both used to demonstrate their discrepancies in various performance comparisons even though they have the same ILP classification.

IV. Motivation

In a typical SMT system, a shared IQ (Issue Queue) usually is used as a set of centralized reservation stations, for all functional units. When an instruction from a thread is dispatched from its corresponding ROB, it is then assigned an entry in the IQ waiting for its source operands to become ready and an available functional unit for it to issue to. Each entry in the IQ (or the corresponding reservation station in the traditional model) is required to hold all the necessary information for the instruction, including the contents of all source operands, before it can be issued. Thus, further increasing the number of entries in the shared IQ usually is hampered by the cost factor, and therefore the utilization of these limited resources becomes very critical to the overall system performance.

The proposed technique is based on the conjectures that, if IQ is not properly managed, its imbalanced occupancy scenario among threads does exist and, it is also likely that threads that are dominating IQ slots may frequently become much less active or even stall. This section will be devoted to the discussion to support these conjectures.

A. IQ Occupancy Analysis

We first look into how exactly the IQ may be dominated by a single thread among the threads in SMT processing. In one of our analytical simulations, Figure 2 shows the percentage of time that at least one thread is occupying at least the given number of IQ entries. \( R \) is used to denote the number of ROB entries and \( q \) is the size of IQ used. When ROB size is relatively larger than the IQ size (as shown in the figure with \((R, q) = (128, 32)\)), the dominance tends to be more severe due to more available instructions for dispatching from a thread. For example, the results from running Mix 3, in over 50% of time there is at least one thread occupying 27 out of the 32 IQ entries, and in over 80% of time at least one thread occupies at least half of the entries, clearly indicating the imbalance of the resource usage. Such a single-thread usage dominance may easily lead to performance degradation when the dominating thread suddenly stalls leaving very few precious entries for other threads to compete for. For the mixes (mixes 2 and 4) that are not as imbalanced, still in roughly about 40% of time at least a thread is taking half of the resources. Figure 3 shows the average percentage among all 6 mixes. This clearly shows that on the average in over 60% of clock cycles there is at least one thread occupying more than 50% of IQ slots, and three quarters of resources are tied up by one thread in over 30% of time. When such a dominating thread somehow becomes “inactive”, the consequence can be significant.

<table>
<thead>
<tr>
<th>Mix</th>
<th>Benchmarks</th>
<th>Classification (ILP)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Low</td>
</tr>
<tr>
<td>Mix 1</td>
<td>perlbmk, mesa, swim, crafty</td>
<td>1</td>
</tr>
<tr>
<td>Mix 2</td>
<td>mcf, equake, vpr, lucas</td>
<td>4</td>
</tr>
<tr>
<td>Mix 3</td>
<td>appla, ammp, mgrid, galgel</td>
<td>0</td>
</tr>
<tr>
<td>Mix 4</td>
<td>mcf, equake, mesa, crafty</td>
<td>2</td>
</tr>
<tr>
<td>Mix 5</td>
<td>perlbmk, lucas, galgel, gcc</td>
<td>1</td>
</tr>
<tr>
<td>Mix 6</td>
<td>mesa, swim, apsi, mgrid</td>
<td>1</td>
</tr>
</tbody>
</table>

**TABLE II. Simulated Multi-threaded Workload**
The inactiveness percentage goes up to 90% when a thread takes up at least 19 of the 32 IQ slots. For example, in very prevalent situations that threads become inactive when occupying a large amount of IQ slots. This result clearly shows that all mixes have been issuing any instruction for at least 10 consecutive clock cycles when a thread becomes “inactive”. A very lenient condition is given here to classify a thread to be “inactive”: when a thread has not been issuing any instruction for at least 10 consecutive clock cycles. This result clearly shows that all mixes have very prevalent situations that threads become inactive when occupying a large amount of IQ slots. For example, in Mix-2 when a thread takes up at least 19 of the 32 IQ slots over 80% of time it is in the “inactive” mode as defined. The inactiveness percentage goes up to 90% when a thread occupies over 75% of slots. In general, when the per-thread dominance is more prominent, the inactiveness likelihood becomes higher. This scenario undoubtedly will block the flow of other threads which can only compete for the leftover of the IQ slots.

Another simulation is performed to look into the likelihood that a thread becomes inactive when already occupying a significant amount of the IQ slots. Figure 4 shows, for each of the six mixes, out of all clock cycles when a thread is taking up at least a specified number of IQ slots the percentage of clock cycles that this thread becomes “inactive”. A very lenient condition is given here to classify a thread to be “inactive”: when a thread has not been issuing any instruction for at least 10 consecutive clock cycles. This result clearly shows that all mixes have very prevalent situations that threads become inactive when occupying a large amount of IQ slots. For example, in Mix-2 when a thread takes up at least 19 of the 32 IQ slots over 80% of time it is in the “inactive” mode as defined. The inactiveness percentage goes up to 90% when a thread occupies over 75% of slots. In general, when the per-thread dominance is more prominent, the inactiveness likelihood becomes higher. This scenario undoubtedly will block the flow of other threads which can only compete for the leftover of the IQ slots.

B. Inactive Resource-Dominating Threads

Another simulation is performed to look into the likelihood that a thread becomes inactive when already occupying a significant amount of the IQ slots. Figure 4 shows, for each of the six mixes, out of all clock cycles when a thread is taking up at least a specified number of IQ slots the percentage of clock cycles that this thread becomes “inactive”. A very lenient condition is given here to classify a thread to be “inactive”: when a thread has not been issuing any instruction for at least 10 consecutive clock cycles. This result clearly shows that all mixes have very prevalent situations that threads become inactive when occupying a large amount of IQ slots. For example, in Mix-2 when a thread takes up at least 19 of the 32 IQ slots over 80% of time it is in the “inactive” mode as defined. The inactiveness percentage goes up to 90% when a thread occupies over 75% of slots. In general, when the per-thread dominance is more prominent, the inactiveness likelihood becomes higher. This scenario undoubtedly will block the flow of other threads which can only compete for the leftover of the IQ slots.

V. Capping Per-Thread IQ Usage

There are many different ways of controlling the usage of IQ for the threads. One straightforward approach is to simply cap the number of slots each thread is allowed to occupy at any time. Let $q$ and $n$ denote the size of IQ and the number of threads, respectively. Obviously any cap value below $q/n$ will not lead to any performance gain, since with a cap value $c < q/n$ there will be at least $q - n \cdot c$ ($> 0$) slots constantly left unused, which is essentially the same as using a smaller IQ. Thus, such a cap value should never be set at lower than $q/n$, and the performance with a cap value $c_1$ will be always worse that with a cap value $c_2$ if

$$c_1 < c_2 < q/n$$

(1)

For a cap value $c \geq q/n$, overall performance may vary depending on transient behavior of threads. In general, if the cap value is set too high, the benefits of allowing other threads to use the IQ when the dominating thread becomes idle will not be as prominent. On the other hand, if the cap value is set too low, overall performance may suffer if the amount of “concurrent” activities among threads is not high; that is, not all threads are active at the same time and thus utilization of IQ is sacrificed by setting a low cap value. Another argument for a low cap value stems from the situation when the thread ILP is low (thus out-of-order issuing and execution is not prevalent), in which allowing for a higher cap value does not benefit thread throughput much, if any.

Plots in Figure 5 show the IPC (Instructions per Clock Cycle) values for different cap values. The combinations
of ROB size and IQ size tested are \((R,q) = (128,32), (96,32)\), and \((64,32)\). As aforementioned, any cap value below \(8 (= q/n)\) will lead to detrimental effects, which is clearly demonstrated in these figures. The claim about the monotonously decreasing performance in Equation 1 when the cap value is reduced below the threshold of \(q/n\) is also clearly verified. Note that when the cap value is set at 32 \((= q)\), it becomes the original default configuration without any cap restriction.

In order to further illustrate how different cap values would affect the performance of different mixes. Plots in Figure 6 and Figure 7 give the IPC comparison for each individual mix with cap values set no lower than 8 using the combination of \((R,q) = (128,32)\). From these results, we can easily conclude that for some mixes the proposed capping approach has a very significant impact on performance; however, different mixes have their performance peak at different cap values. For example, for mix 2, the performance gain peaks at the cap value of 13
with a margin of performance gain close to 9%, and mix 5 peaks at a very small cap value of 11 with a performance gain close to 7%. For some other mixes, the gain is not as significant and most of the time for these mixes the peak performance happens at a cap value very close to the q, indicating the less effect from such a method.

Since it is impossible to prescribe a different cap value for a given mix in advance in order to achieve the best performance, either a fixed cap value has to be issued or some more intelligence needs to be incorporated for an adaptive approach. Here we first show the performance gain from a fixed cap value. Figure 8 shows the average performance gain among all 6 mixes when a fixed cap value is used. Again here only cap values of at least 8 are used. When ROB sizes are smaller compared to the IQ size, for example R = 48 and R = 64, the best cap values are toward the lower range – R = 48 peaks at cap value of 13 and R = 64 peaks at 15. On the other hand, when ROB sizes are relatively larger compared to the IQ size, the best cap value tends to move up to the higher range – R = 96 peaks at 23 and R = 128 peaks at 25. In general, from these mixes, if a cap value is set at anywhere between 40% and 75% of the IQ size, a very dependable performance gain can be expected.

Compared to other adaptive dispatching algorithms which require extensive additional storage hardware and control logic that may delay real-time processing, this technique achieves a very respectable performance gain with very little extra logic – a few extra counters for retaining thread IQ occupancy information and a few comparators for checking against the set cap value.

VI. Adaptive Capping

From the previous section, the capping approach does show very promising results while an arbitrarily assigned fixed cap value does not always lead to a desirable gain for different mixes. It remains to see if the cap value can be dynamically adjusted for different mix situation, or even, for the same mix, be adjusted according to different real-time transient behaviors demonstrated by each individual program.

The proposed adaptive capping method is based on a notion that the more threads that are currently “actively” moving through the system, the sharing of IQ slots among
threads should be enforced more; while on the other hand, if there are fewer threads currently “active”, then each thread should be allowed to occupy more IQ slots. This leads to a simple conjecture that the cap value should be dynamically adjusted according to the number of “active” threads at that time.

There are two issues to address in order to effectively implement this approach: how to dynamically classify “activeness” of a thread, and how to set the corresponding cap values for different activeness scenario, that is, different number of active threads. For the “activeness” issue, we choose to set a time threshold – if a thread is not issuing for a certain consecutive number of clock cycles, then it is declared to be “inactive”. Note that true activeness of a thread should always be a relative degree-wise quantification instead of a boolean classification used here. Such an overly simplified classification is adopted for the sake of fast processing and minimal hardware requirement. The value of this setting, the consecutive number of clock cycle a thread has not been issuing (denoted as \( cc_{nis} \)), obviously will affect the performance dearly since it directly determines how likely a thread is to be declared “inactive”. If this number is set too small, then it becomes more likely that more threads will be declared “inactive” which leads to higher cap values, which in turn may jeopardize the chances for some of the threads that are actually active to dispatch due to high cap values. On the other hand, if this threshold is set too high then it becomes less likely for a thread to be declared “inactive” (and with a longer delay before a truly inactive thread to be declared so), leading to in general smaller cap values, which may restrict truly active threads from fully utilizing the IQ while waiting for the inactive threads to be declared “inactive”. The number of “active” threads thus classified is denoted as \( n_{active} \).

The second issue is on the exact setting of the various cap values (denoted as \( n_{cap} \)) for different number of active threads. A systematic formula is adopted here in our proposed technique to dynamically set this cap value:

\[
n_{cap} = q \times (1 - n_{active} \times d)
\]

where \( q \) is the total size of IQ and \( d \) represents a set decrement percentage to further reduce the cap value for each additional currently “active” thread to promote the sharing of IQ among the “active” threads. For example, if \( q = 32 \) then the the cap values will be dynamically set according to Table III under different \( d \) values. Note that cap values are supposed to be integral numbers only; thus, the cap values shown in the table for \( d = 15\% \) are essentially treated as 32, 27, 22, 17, and 12, respectively. For the purpose of a simple illustration, this dynamic setting follows a straightforward linear decrement percentage with each additional “active” thread; however, a nonlinear decrement scheme is likely to lead to a better overall setting, which will not be discussed in this paper due to its arbitrariness.

Setting of this decrement percentage value, \( d \), is also critical to an effective scheme. If this percentage is set too small, the difference between caps values used for different number of active threads becomes too small for the benefits of the capping method to realize, while, on the other hand, if this decrement is set to high, utilization of IQ may easily suffer if activeness of threads is not declared properly. That is, there exists a strong correlation between the setting of \( cc_{nis} \) and the setting of \( d \) to lead to an effective approach.

In our simulations, a spectrum of \( cc_{nis} \) values from 1 to 20 are tested, while a set of three different \( d \) values are used: 5%, 10% and 20%. All 6 mixes are tested individually (see Figure 9 and Figure 10). We can see that different mixes of benchmark programs react very differently to the combinations of \( cc_{nis} \) and \( d \). Some mixes, such as Mixes 2, 4 and 5, prefer a larger decrement percentage value, all performing better with \( d = 20\% \), while the other three mixes (1, 3, and 6) under-perform with \( d = 20\% \) than with other smaller \( d \) values.

Average improvement from this technique on the 6 mixes is shown in Figure 11. Compared to the fixed-cap approach in the previous section, where with \((R, q) = (128, 32)\) the best improvement lingers around 2%, this adaptive approach further raise the performance to close 3% when the correct \( cc_{nis} \) and \( d \) are selected. Again, such an improvement does not require a significant increase in extra hardware – a few additional counters and comparators will suffice.

**VII. Conclusion**

This paper clearly demonstrated that utilization of resources shared among the threads in an SMT system could significantly affect the overall performance. By simply capping the resource usage of any thread, utilization of the
critical resources can be vastly improved. Further adjusting such a cap according to real-time thread behaviors again demonstrates additional performance gain. A nonlinear setting of the cap values should further improve the potential by allowing a more flexible cap adjustment when the size of IQ is changed. An autonomous adjustment scheme will be even more appealing to allow the system to self-adjust the cap value to reach the best performance by monitoring the transient behavior of the threads. These will be further studied in the future research.

References

Fig. 11. Average IPC Improvement with Various $d$ Values Using Adaptive Capping

Tilak K. D. Nagaraju
Tilak Kumar Devalapura
Nagaraju received his B.S. degree of Engineering in Electronics and Communications in 2005 from New Horizon College of Engineering, Bangalore, India. He then received his M.S. degree in Electrical Engineering at the University of Texas at San Antonio in December 2008 and an M.S. degree in Computer Engineering also from UTSA in August 2011.

Caleb Douglas
Caleb Douglas is a Senior currently pursuing his B.S. degree in Electrical and Computer Engineering at the University of Texas at San Antonio. Caleb was a member of the NSF-REU funded ESCAPE (Experimental Study on Computer Architecture and Performance Evaluation) program at UTSA 2011. He plans to graduate in Fall 2012 then either pursue a M.S. in Computer Engineering or enter the industry. His research interests are in Computer Architecture, specifically CPU Performance Evaluation.

Wei-Ming Lin
Wei-Ming Lin received the BS degree in Electrical Engineering from National Taiwan University, Taipei, Taiwan, in 1982; the M.S. and Ph.D. degrees in Electrical Engineering from the University of Southern California, Los Angeles, in 1986 and 1991, respectively. He was an assistant professor in the Department of Electrical and Computer Engineering at Mississippi State University before joining the University of Texas at San Antonio (UTSA) in 1993, and, since 2004, he has been a professor of Electrical Engineering there, and also the Associate Dean for Graduate Studies in the College of Engineering since 2006. His research interests include distributed and parallel computing, computer architecture, computer networks and internet security.

Eugene John
Eugene John received his Ph. D. in electrical engineering from the Pennsylvania State University. He is currently a Professor in the Department of Electrical and Computer Engineering at the University of Texas at San Antonio. His research interests include Low Power VLSI Design, Computer Architecture, Multi-core Systems, Nanoelectronic Systems, Computer Performance Analysis, Multimedia and Network Processing and Reconfigurable Computing.