Automatic Phase Detection and Adaptation for an Asymmetric Multi-Core Environment

Matthew Beckler, Vinod Chandrasekaran
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213
{mbeckler, vinodc}@cmu.edu

May 10, 2009

Abstract

Automatic program phase detection is used to characterize threads of execution. Using various performance metrics such as *memory-level parallelism* and *instruction-level parallelism*, a program thread can be assigned to the most appropriate processor core in an asymmetric multi-core environment.

1 Introduction

With many computer architects investigating the use of asymmetric multi-core processors, an important question arises: *How do we perform optimal scheduling of threads in an asymmetric processing environment?*. Current scheduling concepts assume that all processing elements are equal in ability and quality, and use time as the primary resource. These simpler ideas would certainly work in an asymmetric environment, but often a thread of execution could be limited because it is running on a "slow" core, or a non-demanding thread could be wasting hardware resources by running on a "fast" core. By characterizing each thread's behavior, we can find an optimal assignment of threads to cores, increasing overall performance and decreasing unnecessary power consumption and resource contention.

2 Background and Related Work

Throughout the history of the microprocessor, Moore's law has "provided" for a doubling in the number of transistors per chip roughly every 18 months. Along with this exponential growth in the number of transistors, we have also seen comparable rates of decrease in minimum feature size and increase in core processor frequency. In the past 10 years, the increases of chip frequency appear to have stalled, with chip power and power density limiting further increases in clock speed. With increases in chip fre-

quency no longer able to satisfy the demand for increased performance, new alternatives must be considered.

2.1 Chip Multi-Processors

Microprocessor vendors have started using all the extra transistors to create multiple copies of a processor core on the same chip, a concept called CMP (Chip Multiprocessor). Traditionally, in a CMP setup, all of the processor cores on a chip are exactly the same, which simplifies the design and test resources required to get a product to market. This symmetric multi-core situation is ideal for programs that are highly parallelizable, but most if not all real-world applications have serialized sections that can not be parallelized. These important serial sections must be run on one of the cores, while the other cores sit idle waiting for the parallel section of code.

One possible solution to this problem would be to change the multi-core setup away from having multiple identical cores on one chip, and instead have different types of cores on each chip. This setup could be called an Asymmetric CMP, and would retain the multi-threaded performance of a symmetric CMP, but would also provide improved single-thread performance for serialized sections of program execution. Figure 1 shows three examples of CMP chips.

2.2 Phase Detection

As a program runs on a processor, it typically passes through different phases of execution. Each phase may have different performance characteristics, and therefore different hardware resource requirements. If we could detect the presence of significant phase changes during the execution of an application, we could adapt to the different performance characteristics by re-allocating the processor cores most suitable to the current phase. The would allow us to optimize processor performance, power usage, and resource availability dynamically at runtime.

Dhodapkar and Smith have a very good survey paper comparing existing phase detection techniques [1]. One very basic phase detection technique from the same authors [2] is the idea of tracking a program's working set over an interval as the set of instruction addresses that were used over the course of a window of execution. They define the distance between the working sets of two adjacent sampling intervals (i and $i-1$) as:

$$\Delta_{i,i-1} = \frac{||W_i \cup W_{i-1}|| - ||W_i \cap W_{i-1}||}{||W_i \cup W_{i-1}||}$$

Dhodapkar and Smith also developed a "working set signature", basically a hashing function for instruction addresses, to have a concise characterization of each working set, allowing for easy comparisons between adjacent working sets. They found that a signature size of as few as 32 bytes was sufficient to obtain good results in phase detection.

A technique involving BBV (Basic Block Vectors) was introduced by Sherwood, Sair, and Calder [3]. They defined a BBV to be a set of counters, each of which counts the number of times a static basic block is entered in a given execution interval. The BBV distance between sampling intervals (i and $i-1$) is defined as:

$$\Delta_{i,i-1} = \sum_{j=0}^{\infty} |\text{count}_{i,j} - \text{count}_{i-1,j}|$$

3 Implementation

3.1 Phase Metrics

To simplify things, we will be characterizing program phase behavior in terms of two met-

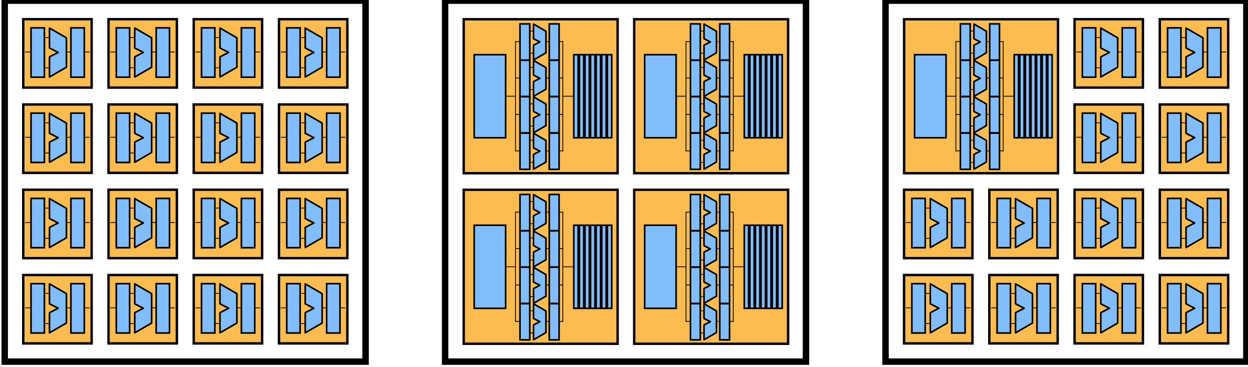

Figure 1: **Different types of multi-core chips.** (left) 16 small cores in a symmetric multi-core chip. (center) 4 large cores also in a symmetric multi-core chip. (right) An asymmetric multi-core chip with one large core and 12 small cores.

rics, MLP (*memory-level parallelism*) and ILP (*instruction-level parallelism*):

ILP. When many nearby instructions in an instruction stream are relatively independent of each other, meaning they do not have many data dependences between them, we can say that this area of the program shows a high level of ILP. This means that many of these instructions can be issued and executed in parallel to gain a performance boost. An example of a program phase with high ILP would be a vector array operation, such as iteratively adding two arrays' elements together and storing the result into a third array, where each loop iteration is completely independent of the previous iteration. On the other hand, a program phase that has many instructions "chained" together with data and control dependences is considered to have a low degree of ILP. Here, a classic example is linked-list traversal, where each node of the list must be examined in order to find the pointer to the next node in the list. Even if the code is in a loop, it is no faster and no more parallelizable than straight-line dependent code.

Program phases that have low levels of ILP would not gain any benefit from running on an out-of-order (OoO) processor core, because many instructions would be stalled while waiting for previous instructions to finish. Similarly, a superscalar core would also not provide much of a speedup compared to an in-order core, because of the difficulty in finding multiple independent instructions to be issued simultaneously. In such cases, it would be a good idea to migrate these threads to simpler and more power efficient in-order cores. Ideally, there would be very little difference in observed performance for this thread whether running on the in-order or out-of-order core. Switching to the in-order core would also potentially free up the use of the relatively "nicer" out-of-order core for another application thread, with characteristics more suited to exploit the features of the core.

We earlier made the statement that it is preferable to run a serial section of code (such as linked-list traversal) on a more powerful core. When we measure the level of ILP in a program phase, does a high degree of ILP point to the fact that the code can be run in parallel across several in-order cores, or that it would benefit from running on a powerful core? In our work, we are concerned only with single-thread performance, so a section of code in a thread with a high-

level of ILP would benefit from being run on a powerful core. It is upto the user to write parallel code in a multi-threaded fashion such that it can be distributed across cores.

MLP. In modern microprocessors, the latency of the memory system is many orders of magnitude slower than the processor and L1 cache. Some cache misses can take several hundreds of clock cycles to be serviced, causing a tremendous amount of latency and wasted time in the processor. Some processors have the ability to service multiple outstanding cache misses concurrently, and for these processors, the amount of memory-level parallelism can be an important factor in the overall performance. If this type of processor is executing a program and processes two nearly-adjacent memory instructions that cause cache misses, then the long delay time for the memory accesses can be overlapped, nearly halving the amortized stall time. With more adjacent cache misses, the average memory access time drops further. If a program phase has memory misses that are spread out so that they cannot be overlapped, then the phase has a low level of MLP.

In most microprocessors there is the idea of an *instruction window*, that consists of all instructions that are "in flight", as they progress from decoding and dispatch, executing in the functional units, and finally re-ordering and in-order retirement at the back-end of the processor. An instruction window helps to keep track of the instructions' ordering and dependencies, and partially enables out-of-order execution. For processors that have a small instruction window, a single long-latency cache miss can cause the instruction window to fill, causing full-window stalls until that memory request is satisfied. A larger window would enable more flexibility in handling cache misses. If a program phase that is executing on a powerful core (with a large instruction window) has a low level of MLP, meaning that the memory misses are not grouped together, then a core with a smaller instruction window can be used without much compromise in performance. This is because the average number of stall cycles caused by misses would not be appreciably worse, irrespective of the size of the instruction window of the core it is run on.

3.2 Software Implementation Details

For this research project, we used a full-system, timing-accurate simulator setup comprising Virtutech's Simics functional system simulator and the SimFlex project's Flexus architectural models. Being a full-system simulator, Simics can simulate a complete computer system, including main memory, hard disks, and other peripherals. It allows us to run benchmarks that require system libraries or operating system support. The SimFlex project at CMU created the Flexus architectural models as a plugin to Simics to allow for more accurate simulations of the processors' internal behavior. Currently, the official Flexus distribution only supports multi-core systems that are symmetric, with all processor cores defined identically by the same configuration objects. We wanted to investigate asymmetric multi-core environments, which required some modifications to the Flexus codebase. Figure 2 shows both the original, unmodified Flexus symmetric CMP simulator `CMPFlex.OoO` as well as the extended and modified `CMPFlex.OoO.Asym` asymmetric CMP simulator, which will be described in more detail in section 3.3.

The `CMPFlex.OoO` processor simulator model is an out-of-order, multi-core capable

simulator. Each core is actually defined as five different sub-components:

- μArch - The microarchitecture
- μFetch - The L1I cache
- Fetch Address Generate - Fetch and prefetch engine
- v9Decoder - Instruction decoder
- L1d - L1 data cache

Each of these components is separately configurable, lending good flexibility to the system. In order to move away from the identically array-instantiated situation, we started by separately instantiating each of these five components for each core of our desired asymmetric system. This broke a large number of assumptions in the Flexus code base, and required a large amount of clean-up and sanitizing code to try and mold Flexus into behaving in a new way. This was not very efficient or particularly informative, and was not 100% error-proof. One major source of trouble was the breakage of the automatic memory connections and routing. For example, port definitions are used to define how one module connects to another module, and can be either a single port or an array of ports. Unfortunately, the array of ports stops working when using separately instantiated components, so we had to create a new Flexus module to arbitrate between the individual cores and L2 cache.

3.3 Components Created

MemoryMux - As mentioned above, we had to create a module to deal with memory cache connections. We called this module the MemoryMux, and it was placed between the L1 caches in the L1D and μFetch and the L2 cache connected to the memory system. Despite being a relatively simple component, its construction taught us a lot about the internal workings of Flexus' memory transaction model and the differences between single ports and array ports.

MLPListener - In order to be able to detect the amount of memory-level parallelism in the current program phase we created a new module that sits between the L2 cache and the memory subsystem. Any memory accesses that miss in the L2 cache will be forwarded through the MLPListener module, and passed on to the memory system. The MLPListener filters and tracks the number and distribution of L2 misses, and tries to compute a good metric for MLP. For a given window size (such as 1000 cycles), the module keeps a counter of the total L2 misses observed. It also tracks the cycle number of the most recent miss, used to determine the distribution of misses. Based on experimentation and observation, we determined that the miss fulfillment latency for main memory was approximately 265 cycles. Using this value, we were able to roughly compute whether or not a given L2 miss was independent of the previous L2 miss.

Once we have a count of the total and independent L2 misses, we can plot their values, such as in Figure 3, for the benchmark SPEC2K-ART. An important enhancement we can do is to take the ratio of the "nearby" or overlapping misses to the total misses. This produces a normalized value between 0 and 1, and makes the MLP metric have the same range and interpretation regardless of processor or workload details.

ILPListener - While the MLP metric can be computed by listening to the L2 cache misses, the ILP metric requires different information. Some of this extra information includes the instruction dependencies, which are most easily available at the retirement stage of the processor, as each instruction is

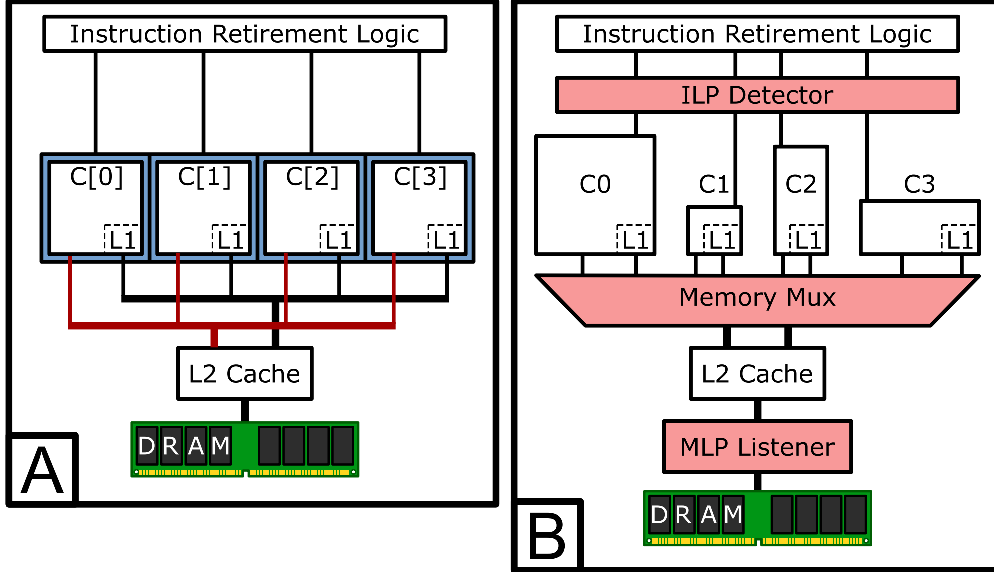

Figure 2: **Flexus Processor Architectures** (A) The original array-instantiated symmetric processor `CMPFlex.OoO`. (B) The extended asymmetric `CMPFlex.OoO.Asym` processor, with newly created components highlighted in red.

removed from the ROB (re-order buffer).

4 Experimentation and Results

Since Flexus is unable to perform thread migration, all of our experiments were performed with single-threaded workloads. Since we are striving to create metrics that are microarchitecture-independent, we believe that using single-threaded workloads is fair, and the results should remain valid when considering multiple threads running on all cores.

4.1 MLP Detection

As mentioned above, the `MLPListener` module was situated between the L2 cache and the main memory system. The module listenes to the L2 misses, and uses their frequency and distribution to compute the overall MLP metric. As an example, we ran the "SPEC2K Art" benchmark, with results in Figure 3. The left plot shows the total number of L2 cache misses and what portion of those misses were non-overlapping misses. We can see that this particular benchmark has an initial period of high memory usage, with very few non-overlapping misses, meaning that nearly all of these misses were very close in time, and would be a good fit with some advanced core's memory parallelism features. Once this phase of high memory use is finished, the benchmark transitions to a low memory usage phase, with nearly no L2 misses present.

In looking at the left plot of Figure 3, we see that the values produced are dependent on the both the workload and the sampling interval. This is not ideal, as it would give us difficulties when trying to find the best thresholds, so we take the ratio of overlapping misses to total misses in each interval. This produces a value that is normalized between 0 and 1, and is independent of sampling period and the specific workload's total

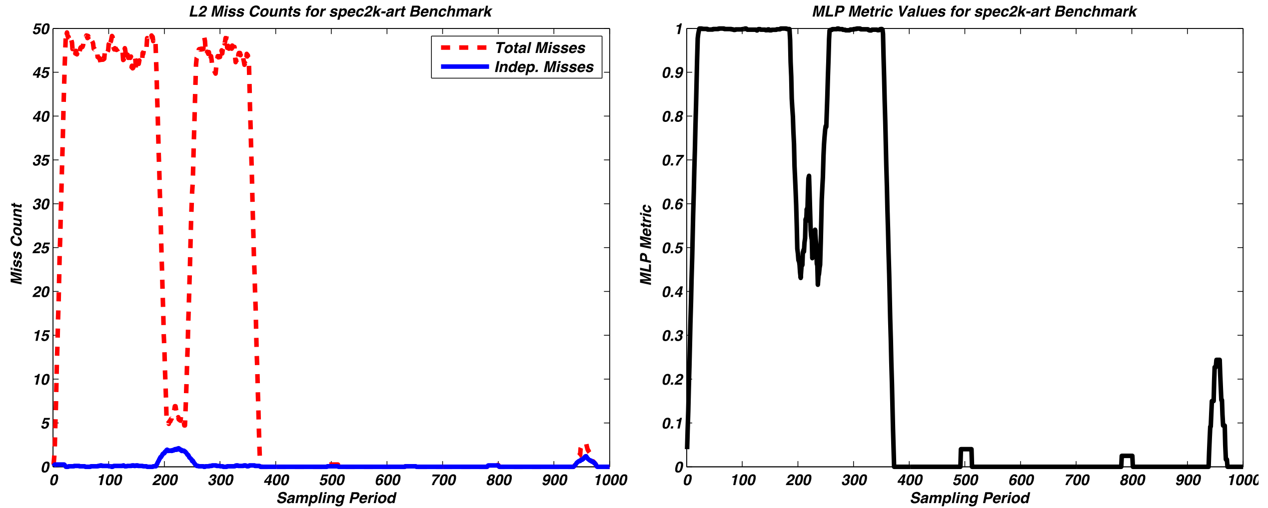

Figure 3: **MLP Detection -** (left) Raw counts of non-overlapping and total misses for SPEC2K-ART. (right) MLP metric of overlapping misses divided by total misses. Note that this metric produces a value normalized between 0 and 1. Plots were smoothed with a 20-element weighted moving filter.

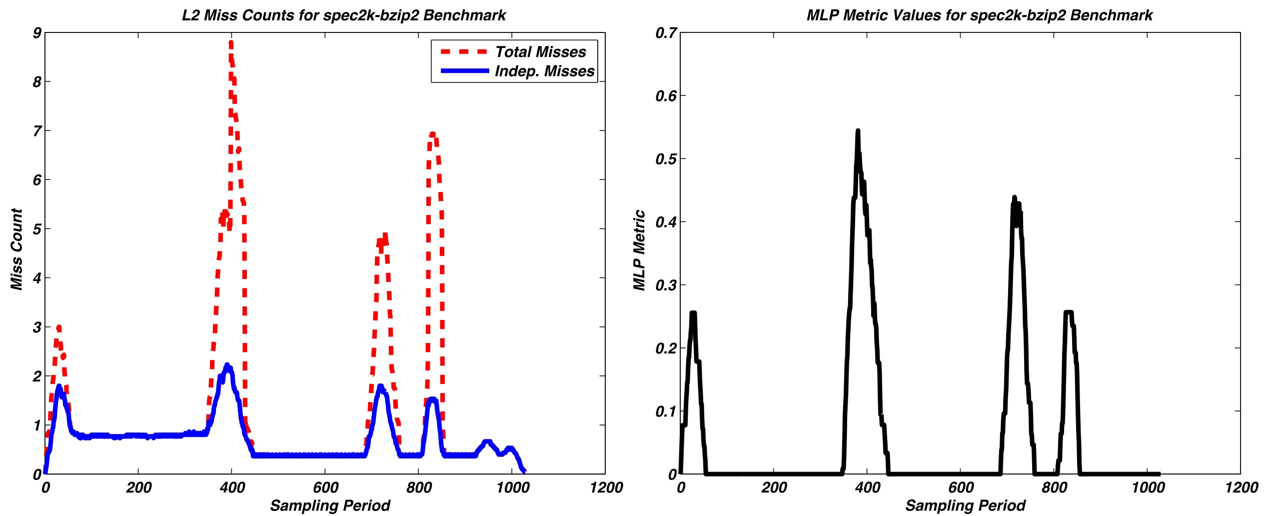

Figure 4: **MLP Detection -** (left) Raw counts of non-overlapping and total misses for SPEC2K-BZIP2. (right) MLP metric of overlapping misses divided by total misses. Note that this metric produces a value normalized between 0 and 1. Plots were smoothed with a 20-element weighted moving filter.

memory usage.

When we run the SPEC2K-BZIP2 benchmark, we get the data shown in Figure 4. This benchmark has several stable areas and several disorderly areas. We can see that the metric never settles down enough to cause a switch to a different core, and stays on the core for low MLP threads.

Once we have a metric for MLP, we can start using it in scheduling decisions. For

this example the thread can be scheduled for an out-of-order core with a large instruction window, in order to effectively handle and amortize the memory delay. Between the two periods of high memory usage is a short respite with lower MLP, and we should set our timeout threshold to be longer than this (and other similar delays in other benchmarks) in order to avoid switching cores. Only when we have detected a longer period of low MLP should we reschedule this thread to a core with a smaller window, as the large window is no longer an advantage without the intensive memory accesses. Also notice that in the right plot, there is some noise in the later values, producing a few bumps in the MLP metric plot. These values should also be ignored, as they are not indicative of a major phase change.

5 Conclusion

While the Flexus simulator was not able to handle thread migration, we were still able to investigate different metrics for characterizing phases of program execution. Detecting memory-level and instruction-level parallelism was useful and viable for determining program phases, and gives information valuable to the thread scheduling process.

6 Usage Instructions

Here are details and instructions on using our modified version of Flexus with added modules. A complete distribution of Flexus, based on the 18741 release version is included in the attached tarball. The original `CMPFlex.OoO` simulator was copied and modified to become `CMPFlex.OoO.Asym`. The three new modules are included in the `/components/` subdirectory.

Build Instructions - From the top-level directory, simply type `make CMPFlex.OoO.Asym` to build the code for this modified simulator. If you have a multi-core machine, you can use `-j N` to enable multi-threaded compilation with N concurrent threads.

After the code is built, we have included a short run script called `run_job.sh`, which runs the SPEC2K-art benchmark. The MLP detector will print out messages relating to MLP metric stability, and notices when the phase settles down and would be a good time to re-evaluate the core assignments.

For more information about the modified codebase as well as building and running instructions, please see the file `18741-README.txt` file inside the code distribution directory.

7 Future Work

We were recently told that even with extensive modifications, Flexus is still unable to do thread switching without crashes and lockups, likely due to communications between caches (probably the snooping and coherency messages). To continue this research, therefore, it would be most prudent to switch away from using Flexus/Simics as the simulation platform, and instead use a different simulator. Professor Mutlu has mentioned a couple of other simulators that might be viable for these ideas, and we may investigate them in the future.

Through the course of this project we learned a great deal, both about this topic in computer architecture, but also about working with large, legacy codebases. "Get help early, and get help often," was an idea we tried to follow, and the course professor and teaching assistants were indeed very helpful when we were struggling to figure out some of the tricky and confusing features of Flexus. We learned that it doesn't work very

well to get hung-up on infrastructure, even if infrastructure changes are a majority of the project. We learned that documentation in all forms, from low-level function and method comments, to high-level class and module documentation, is a critical part of software development, and invaluable when others may need to read, understand, and modify the codebase. Our experiences with the Flexus code base have given us new resolve to never produce undocumented software, lest other poor souls be left to wander through a similarly confusing and undocumented project.

If the major issues with the simulator environment are worked out, then there are some other interesting areas of work that would be good to investigate. Expanding the project scope to other metrics than MLP and ILP would be an interesting area of exploration, especially if it was possible to find metrics that lined up more closely with the available processor core characteristics. Another area of interest would be to evaluate the actual microarchitecture independence of all metrics, to ensure that they work just as well regardless of the currently used core.

8 Acknowledgements

We would like to thank course professor Onur Mutlu and the course Teaching Assistants for their thoughts, help, advice, suggestions, bug fixes, code samples, and general tips. We also need to thank Ben Nowak and Steve Thompson for their guidance as we worked towards getting Flexus ready to handle thread migration, and for letting us know that it isn't really possible to do thread migration using Flexus without first rewriting a large part of the simulator. Eric Chung of the ECE CALCM Group also helped us with accounts and settings for the Scotch cluster, which was very useful for running Flexus simulations.

9 References

[1] Dhodapkar, A. and Smith, J. E. “Comparing Program Phase Detection Techniques”, IEEE/ACM International Symposium on Microarchitecture, 2003.

[2] Dhodapkar, A. and Smith, J. E. “Managing Multi-Configuration Hardware via Dynamic Working Set Analysis”, International Symposium on Computer Architecture, 2002.

[3] Sherwood, T. Sair, S., and Calder, B. “Phase Tracking and Prediction”, International Symposium on Computer Architecture, 2003.

[4] Hill, M.D.; Marty, M.R., “Amdahl's Law in the Multicore Era,” Computer , vol.41, no.7, pp.33-38, July 2008

[5] Nagpurkar, P. et al, “Online Phase Detection Algorithms”, Proceedings of the International Symposium on Code Generation and Optimization, 2006