Predictive Thread-to-Core Assignment on a Heterogeneous Multi-core Processor

Tyler Sondag
Dept. of Computer Science
Iowa State University
sondag@cs.iastate.edu

Viswanath Krishnamurthy
Dept. of Computer Science
Iowa State University
viswa@cs.iastate.edu

Hridesh Rajan
Dept. of Computer Science
Iowa State University
hridesh@cs.iastate.edu

Abstract

As multi-core processors are becoming common, vendors are starting to explore trade offs between the die size and the number of cores on a die, leading to heterogeneity among cores on a single chip. For efficient utilization of these processors, application threads must be assigned to cores such that the resource needs of a thread closely matches resource availability at the assigned core. Current methods of thread-to-core assignment often require an application's execution trace to determine its runtime properties. These traces are obtained by running the application on some representative input. A problem is that developing these representative input sets is time consuming, and requires expertise that the user of a general-purpose processor may not have. We propose an approach for automatic thread-to-core assignment for heterogeneous multi-core processors to address this problem. The key insight behind our approach is simple – if two phases of a program are similar, then the data obtained by dynamic monitoring of one phase can be used to make scheduling decisions about other similar phases. The technical underpinnings of our approach include: a preliminary static analysis-based approach for determining similarity among program sections, and a thread-to-core assignment algorithm that utilizes the statically generated information as well as execution information obtained from monitoring a small fraction of the program to make scheduling decisions.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors - Optimization; D.3.3 [*Programming Languages*]: Language Constructs and Features - Control structures; D.4.1 [*Operating Systems*]: Process Management - Multiprocessing/multiprogramming/multitasking, Scheduling, Threads

General Terms Algorithms, Experimentation, Performance

Keywords static program analysis, heterogeneous multi-core processors, thread-to-core assignment, phase behavior

1. Introduction

The number of processor cores in a die is expected to continue to multiply in coming years making multi-core CPUs with large number of cores a commodity item [6]. The increasing demand for multi-core processors over single core general purpose processors can be attributed to the fact that no significant performance increase is obtained when the processor clock frequency is scaled [14]. Moreover, the processor is mostly idle since the instruction feed rate cannot match the speed of the high performance core. Multi-core processors solve this problem by placing multiple cores on a single die to boost performance and account for better area utilization.

A multi-core processor is heterogeneous when the cores on a single die differ in their characteristics e.g. cache-size, core-type (in-order or out-of-order), clock frequency, instruction issue width, etc [10]. Heterogeneous multi-core processors are seen as a more viable alternative by many due to better die area utilization, reduced heat dissipation, power consumption and often better performance - power ratio compared to their homogeneous counterparts [7]. They also help cater to a wider class of applications with varying levels of parallelism [9].

There are two challenges to efficient utilization of heterogeneous multi-core processors. First, the amount of parallelism present in the workload must increase significantly. This can either be done by automatic extraction of thread-level parallelism [13] or by developing novel parallel programming language constructs [5]. Second, thread-to-core assignments techniques must be developed that match the resource needs of a thread closely with the resource availability, while maintaining fairness and optimizing overall throughput. This paper presents a technique to address the second issue.

The basic idea behind our technique is as follows. A program exhibits phase behavior [15] in that it goes through several phases of execution that show approximately similar runtime characteristics compared to other phases of execution. If we can approximately predict similarity among program phases statically, we can use this information to determine assignments of threads-to-core at a low runtime cost. In particular, by monitoring the execution of an initial phase in the program, we can predict the execution characteristic of the later phases of the program that resemble this phase. Statically predicted similarity and dynamically measured execution characteristics of the representive initial phase can then be used for selecting an appropriate thread-to-core assignment for the later phases in the program.

Based on this idea, our technique consists of a static analyser, which can be augmented into a compiler, and a dynamic monitoring and scheduling framework, which can become part of the operating system's scheduler. The purpose of the static analyser is to classify an application into a set of phases, and to group these phases into clusters such that each phase in the cluster is likely to show similar runtime behavior. The purpose of the dynamic monitoring and scheduling framework is to estimate the *expected* resource needs of a cluster by monitoring the execution of one representative

PLOS '07 October 18, 2007, Stevenson, Washington, USA.

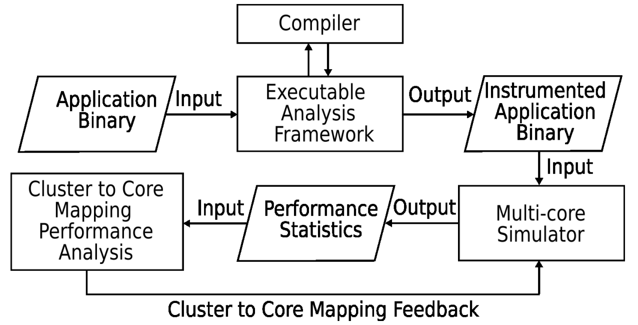
Compiler
Application Binary
Input
Executable Analysis Framework
Output
Instrumented Application Binary
Input
Cluster to Core Mapping Performance Analysis
Input
Performance Statistics
Output
Multi-core Simulator
Cluster to Core Mapping Feedback

Figure 1. Infrastructure

phase in the cluster and to extrapolate the resource needs of the representative phase to determine core assignment policy for other phases in the cluster.

The rest of this paper is organized as follows. Section 2 describes our approach for determining similar phases of execution and mapping phases to processor cores as well as our testing infrastructure in detail. Section 3 presents our planned evaluation framework. Section 4 compares and contrasts our approach with related work. Section 5 describes some potential benefits of our approach and Section 6 discusses future work and concludes.

2. Approach

In this section, we describe our proposed approach for utilizing statically determined approximate program phase information to assist predictive thread-to-core assignment. We first describe our static analysis technique. We then describe our approach for determining the scheduling policy.

2.1 Determining Similar Behaviors

The key requirement for our preliminary analysis approach to determine similar sections in the program is to use only static information about the program unlike other techniques [15] that use dynamic profile of the application. This design decision was made to avoid the need to profile the application, thereby avoiding the need to develop a set of representative test cases for the application.

The technique presented here considers program intervals at the basic blocks level. Basic blocks are then grouped together so that blocks containing similar proportions of instruction types are in the same group. In our approach, basic blocks are grouped into categories of similar basic blocks based on their machine instruction types (arithmetic, data transfer, control transfer, etc.). Currently we only consider instruction types, but a similar technique based on combinations of instructions and operand types may also be considered. We consider only basic blocks larger than a particular number of instructions, which helps us to minimize the overhead associated with frequent core switching that may result otherwise. Currently, we set this number to 20 instructions.

To illustrate, let us consider a basic block which has 70% arithmetic instructions, 10% data transfer instructions and 20% control transfer instructions. This basic block has a majority of computationally intensive instructions. Therefore, this basic block will be grouped with other basic blocks which contain a similar proportion of instructions (i.e. a majority of computationally intensive instructions). The process of creating instruction type vectors and clustering is illustrated in Figure 2. Basic block 1 and 2 and basic block 3 and 4 have similar ratio of instruction types, therefore they are grouped together into clusters.

Our current approach uses a very simple technique that can be significantly improved by considering other information that is also available statically. For example, our current technique does not account for the ordering of instructions that may lead to different characteristics of basic blocks due to control and data dependences between instructions specially when executed on an out-of-order processors. During static analysis, a conservative approach to account for execution order, control and data dependences is likely to lead to better results. Another important factor to account for is cache hits/misses. Two seemingly similar basic blocks may exhibit substantially different behavior depending on the locality of the data accessed. A little more detailed, but still static, analysis that considers the upper bounds on the pipeline stall delays is likely to lead to better approximation. Nevertheless, note that the objective of our static analyser is to provide us with approximate similarity between basic blocks and our initial simple strategy provides a good starting point in this direction.

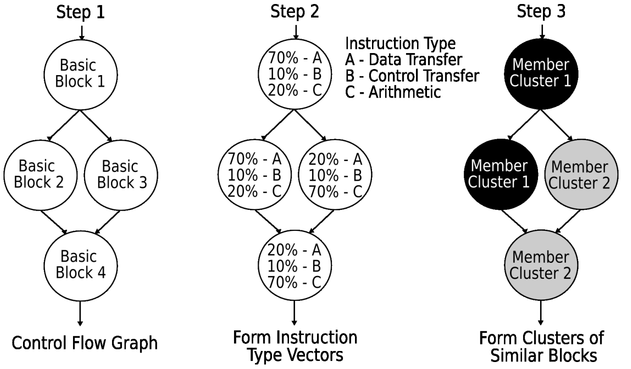

Figure 2. Clustering

1. Randomly select k of the basic blocks, the k vectors corresponding to these basic blocks become the initial cluster centers.
2. For each vector, determine which of the cluster centers is closest and become a member of that cluster.
3. For each cluster, calculate the average of all members of the cluster, this becomes the new cluster center.
4. Repeat steps 2 and 3 until there is no more change in the cluster centers.

Figure 3. k-means algorithm[11]

In the case of the simple example presented in Figure 2 the similarity among basic blocks can be easily identified by exhaustively comparing the vectors of instruction types, but this naíve strategy does not scale for large programs. For classifying similar basic blocks in large programs into clusters, we use the k-means clustering alogorithm proposed by MacQueen [11]. For the benefit of the reader, this algorithm is described briefly in Figure 3.

We apply the k-means algorithm on the normalized vector of instruction types for each block. The normalization is done by dividing the instructions in each category by the total number of instructions within that basic block. Each cluster now contains basic blocks which have a similar proportion of instruction types. Therefore, we may expect each basic block in a cluster to exhibit approximately similar runtime behavior to the other basic blocks in the cluster barring the exceptions that we considered before, which will further narrow down the variance. This method allows us to obtain profile information for any execution path taken at runtime.

We have built our static analyzer using Intel Analysis Tools for Object Modification (ATOM) [1], which serves to demonstrate the feasibility of our approach; however, our approach is not limited to

	Core 0	Core 1	Core 2	Core 3
Cluster A	0	0	-1	1
Cluster B	0	-1	1	0
Cluster C	1	-1	0	0

Performance of cluster type on each core
0 - undetermined
1 - above threshold
-1 - below threshold

Figure 4. Example state of coreResult data structure

the ATOM tool. ATOM is a binary instrumentation engine which contains support for the creation of custom tools for analyzing program structure and collecting runtime information.

2.2 Predictive Thread-to-Core Assignment

Now we have grouped the basic blocks into clusters. Next, we would like to determine an efficient thread-to-core mapping policy. This mapping policy needs to be created dynamically in order to eliminate simulation overhead.

In order to know whether or not a basic block type has previously performed well on a core, we use a data structure to store this information. One such data structure is given in Figure 4.

This data structure is simply a two dimensional array in which the rows correspond to clusters and the columns correspond to the cores. All values are initially set to 0, indicating that we do not know about the execution of that cluster type on that core. Once it is determined that a basic block from a given cluster executes efficiently on a certain core, the corresponding value is set to 1. Similarly, if it is determined that a basic block from a given cluster performed poorly on a certain core, the corresponding value is set to -1.

We use the instructions committed per cycle (IPC) as the measure of poor vs. efficient execution of a basic block on a core. If the IPC executed for a basic block is above a certain (heuristically determined) threshold for this cluster, then the basic block performed well. Otherwise, it did not perform well. In the event that no core executed any basic block type above the threshold for this cluster, the threshold is reduced and the runtime information gathered thus far for this cluster must be reset. This forces the system to gather new information for the new threshold.

```
if(block.ipc > threshold[block.group]){
 coreResult[block.group][getCurrentCore()] = 1
}
else{
 coreResult[block.group][getCurrentCore()] = -1
 decrement = 1
 for each i in numCores{
  if(coreResult[block.group][i] > -1){
   decrement = 0
  }
 }
 if(decrement){
  threshold[block.group] -= stepSize
  for each i in numCores{
   coreResult[block.group][i] = 0
  }
 }
}
```

Figure 5. Finished Executing Block

When a basic block finishes execution on a core, it must be determined how well this core met the resource needs of this workload. An algorithm for this is given in Figure 5.

```
block = current_block
core = -1
if(block.instructionCount > minCount){
 block.group = getGroup(block)
 for each i in numCores{
  if(coreResult[block.group][i] == 1){
   if(coreNotBusy(i)){
    core = i
    break
   }
  }
 }
 if(core == -1){
  for each i in numCores{
   if(coreResult[block.group][i] == 0){
    if(coreNotBusy(i)){
     core = i
     break
    }
   }
  }
 }
 if(core != -1){
  switchToCore(core)
 }
}
```

Figure 6. Beginning Executing Block

When a basic block is encountered that is larger than a certain size, we must determine which core it should execute on. An algorithm for this is given in Figure 6. Using the data structure from Figure 4 check if a core to efficiently execute this cluster type has been determined. If one such core is found, and if the core is not busy, switch the basic block to that core. If no cores have been determined to run this cluster efficiently, then find a core which has not run a basic block belonging to the same cluster as the current basic block, and if it is not busy, switch to that core.

3. Planned Evaluation

In this section, we describe the methods which will be used to evaluate our approach. The infrastructure described here is already setup and evaluation efforts are currently underway. For our experimentation, we are using the M5 Simulator System to execute the application and to determine information about its characteristics at runtime [4]. M5 is a (optionally) full system simulator that currently provides support for different architectures, including chip multi-core processors. M5 simulator supports detailed cycle accurate simulation.

The current configuration that we are using for our test has three slow operating at '1GHz' and one fast processor operating at '2GHz'.

The performance statistics obtained as output from the M5 Simulator System are fed to the performance analysis module which measures the effectiveness of the cluster to core mapping policy. The performance analysis module can instruct the M5 Simulator System to change the current mapping policy, if a mapping strategy which better exploits the heterogeneous core structure was discovered. Using runtime performance statistics we determine how well a basic block belonging to a specific cluster performs on the core to where it was mapped. Once it is determined that a basic block has run effectively on a core, all other basic blocks belonging to the same cluster are executed on that core. The infrastructure to be used in this procedure is shown in Figure 1.

3.1 Workload Construction

Our approach for developing the workload for simulation is similar to that of Becchi *et al.*. They created workloads from 11 benchmarks of the SPEC2000 benchmark suite [3]. Workloads are formed that contain between 1-40 randomly selected threads where each thread corresponds to one of the 11 benchmarks. Each benchmark was run using the M5 simulator to obtain execution traces for each core type available. The output of the M5 simulator helped their CMP simulator (used to model each heterogeneous core configuration) to perform appropriate thread-core assignment. Instead of selecting from only 11 benchmarks, we are considering selecting all benchmarks in the SPEC2000 benchmark suite. We will choose a uniform amount of integer and floating point benchmarks as well as benchmarks which have a large memory footprint and those that do not.

3.2 Evaluation Metrics

We will use weighted speedup to evaluate how well the workloads are executed using our approach [16]. The IPC when all benchmarks in the workload are executed on the fastest core serves as a reference to compare against. This will help us to compare our results with the results of related approaches. Additionally, our results will also be compared with the pseudo best static assignment.

4. Related Work

Since the thread-core assignment policy plays a crucial role in deciding performance, Becchi *et al.* claim that dynamic thread assignment policies better make use of the benefits of a heterogeneous CMP [3]. According to this assignment policy, run-time behavior of threads is observed and thread switching is carried out accordingly. Another phase detection and optimization approach which dynamically optimizes the application based on the current execution profile is given by Nagpurkar *et al.* [12]. Our approach similarly uses dynamic results from the program's execution to change the thread assignment policy.

Kumar *et al.* introduce an approach for dynamically scheduling threads on heterogeneous multi-core architectures [10]. After a certain amount of execution, a trigger is generated which starts a sampling phase which modifies the current thread assignment. Our approach first groups the program instructions into sections statically. Our approach differs from their method in that we monitor the performance of these sections on the assigned core and then apply the performance results to other sections which are determined to be similar.

Sherwood *et al.* presented a dynamic profiling-based technique for similar program phase identification that performs application runs to obtain execution traces of applications [15]. This profile information is then used to identify appropriate simulation points within the application. Here, program execution is broken up into fixed length sections and basic block vector profiles are collected for every section. A similarity metric is used to group similar sections based on their basic block vector profiles. Our approach only uses static information therefore it does not require a dynamic profile.

Our procedure for determining similar basic blocks is also related to the approach used by Sherwood *et al.*, where the clustering is done on the basis of similar intervals of execution [15]. In our approach, basic blocks are grouped into categories of similar basic blocks based on their machine instruction types (arithmetic, data transfer, control transfer, etc.), whereas in their approach Basic Block Vectors (BBVs) are grouped into clusters.

Another related work that strongly motivates the usage of heterogeneous multi-core architectures is by Tullsen *et al.* [8]. This paper shows that heterogeneous multi-core processors offer greater advantages in terms of performance and power when compared with any other class of Chip Multiprocessors (CMPs). The paper stresses the fact that applications achieved performance improvements of up to 40% when power and area constraints were tighter. It also gives us a deep insight into the design issues of a heterogeneous multi-core processor. The paper lays emphasis on tuning cores to a workload set which exhibit certain common execution characteristics rather than running all kinds of workloads. Our approach statically determines which sections of code exhibit similar characteristics in order to tune the thread assignment policy.

5. Discussion

The approach discussed in this paper is likely to provide two major benefits.

5.1 Reducing Profiling and Simulation Overhead

A potential benefit of our approach is reduction in the profiling and simulation overhead of thread-to-core assignment techniques that rely on execution traces to determine appropriate assignments. These techniques often execute a program on a representative test set to determine its runtime characteristics, which is then used to determine appropriate assignments. By monitoring the execution of a few representative phases during the normal execution of the program and then using the results on the fly to determine assignments for later phases, we eliminate the need for a separate profiling and simulation phase.

Moreover, the assignment determined from the techniques that rely on execution traces may become invalid, if the program is executed with a completely different set of inputs, which was not covered during profiling.

Finally, eliminating the need for writing a representative test set for a program is likely to influence the practicality and generality of our approach. We detail some anticipated benefits in the next section.

5.2 Better support for end-users

With the proliferation of multi-core processors in general-purpose computing infrastructure, there is a greater need to find efficient thread-to-core assignment policies for applications developed by end-users. By end-users we mean the group of people who will operate some software system. End-users programming is popularized by languages like Visual Basic that are intuitive enough because of their inherent simplicity that it requires little technical expertise to use it. Another important class of end-users is a typical programmer in a domain-specific language.

As opposed to industry standard benchmarks and off-the-shelf applications, where representative sections (which represent the entire benchmark) can be easily identified and simulated, applications written by end-users might not be frequently run. In addition, end-users are less likely to be computer architecture experts, who can fine tune application behavior based on the architecture of the underlying machine. For example, approaches such as that presented by Balakrishnan *e*t al, require that in addition to the operating system kernel being aware of the asymmetry in the multi-core architecture, the application is also aware of the heterogeneity among the cores in order to obtain a steady performance [2].

End-users are less-likely to rewrite or modify their application in order to better utilize the architectural capabilities. They are also less likely to develop a sufficiently representative test set for profiling and simulation purposes. By eliminating the need for profiling and simulation our technique has the potential to be more useful for programs written by end-users.

6. Conclusion and Future Work

In this paper, we described a technique for thread-to-core assignment for a heterogeneous multi-core processor. Our technique statically determines the approximate phase behavior in a program. This phase behavior and the exhibited execution characteristics of a small set of representative phases is then exploited at runtime to determine likely profitable thread-to-core assignments for later phases of the program. By monitoring the execution of a small representative set, as opposed to simulating the application, our approach is likely to reduce the simulation overhead. With support in the operating system kernels and compilers, our technique can be used automatically without requiring any special input from the user. This may allow programs written by an end-user with no expertise to harness the capabilities of a heterogeneous multi-core processor.

Currently our approach determines similarity at the basic block level. As part of our future work we will group instructions at a more coarse level of granularity. One such approach could be to combine basic blocks into intervals and then form clusters from intervals rather than from basic blocks. This increased granularity will help to reduce the overhead associated with frequent core switching. Increasing this granularity will also give a better picture of the programs execution since the current method considers only certain basic blocks.

Acknowledgements

This work is supported in part by National Science Foundation grant 0627354. The authors would also like to thank the anonymous reviewers for their comments and suggestions.

References

[1] *Intel Analysis Tools for Object Modification (Intel Atom): Release Notes: Release 1.0 Beta*.

[2] Saisanthosh Balakrishnan, Ravi Rajwar, Mike Upton, and Konrad Lai. The impact of performance asymmetry in emerging multicore architectures. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 506–517, Washington, DC, USA, 2005. IEEE Computer Society.

[3] Michela Becchi and Patrick Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 29–40, New York, NY, USA, 2006. ACM Press.

[4] N. L. Binkert, E. G. Hallnor, and S. K. Reinhardt. Network-oriented full-system simulation using m5. In *Sixth Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW)*, 2003.

[5] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM Press.

[6] David Geer. Industry trends: Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005.

[7] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-isa heterogeneous multicore architectures: The potential for processor power reduction. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 81, Washington, DC, USA, 2003. IEEE Computer Society.

[8] Rakesh Kumar, Dean M. Tullsen, and Norman P. Jouppi. Core architecture optimization for heterogeneous chip multiprocessors. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 23–32, New York, NY, USA, 2006. ACM Press.

[9] Rakesh Kumar, Dean M. Tullsen, Norman P. Jouppi, and Parthasarathy Ranganathan. Heterogeneous chip multiprocessors. *Computer*, 38(11):32–38, 2005.

[10] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 64, Washington, DC, USA, 2004. IEEE Computer Society.

[11] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*. University of California Press, 1967.

[12] Priya Nagpurkar, Chandra Krintz, Michael Hind, Peter F. Sweeney, and V. T. Rajan. Online phase detection algorithms. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 111–123, Washington, DC, USA, 2006. IEEE Computer Society.

[13] Jeffrey T. Oplinger, David L. Heine, and Monica S. Lam. In search of speculative thread-level parallelism. In *PACT '99: Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, page 303, Washington, DC, USA, 1999. IEEE Computer Society.

[14] Jeff Parkhurst, John Darringer, and Bill Grundmann. From single core to multi-core: preparing for a new exponential. In *ICCAD '06: Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, pages 67–72, New York, NY, USA, 2006. ACM Press.

[15] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 45–57, New York, NY, USA, 2002. ACM Press.

[16] Allan Snavely and Dean M. Tullsen. Symbiotic job scheduling for a simultaneous multithreaded processor. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 234–244, New York, NY, USA, 2000. ACM Press.