Core Specific Block Tagging (CSBT) based Thread Migration Model for Single ISA Heterogeneous Multi-core Processors

Venkatesh Karthik S[1], Pragadeesh R[1]

College of Engineering, Guindy, Anna University

ABSTRACT

The proposition of a single ISA heterogeneous multi-core architecture as a mechanism for saving power sparked the revolution of experimenting with various thread to core assignment and migration policies. This paper proposes a compiler based approach for migration, which takes into account the ILP inherent in a given instruction sequence (of a thread). In this mechanism, the compiler splits a thread into blocks, tags a block to a core based the underlying core architecture and optimizes the split. During execution, the thread is migrated between cores based on the tags such that power used is minimized without a negative impact on the overall performance. The paper also proposes a mathematical model for the migration policy and derives an expression for the maximum number of parallelizable and non-parallelizable blocks an assembly code could have, up to which migration would be favorable. We have shown by simulations that our migration policy provides better performance/power ratio when compared to assignment of the entire thread to a single core.

1. INTRODUCTION

Heterogeneous multi-core systems consist of cores with varying capabilities and varying levels of power dissipation. This has given rise to a scenario where thread to core assignment critically dictates overall performance and power usage. Various approaches for solving this problem have evolved. The changes in ILP within a thread have severely limited the possibility of the above assignment. Thus the question arises, "What is the optimal core for the execution of a thread or a part of the thread that aims at minimizing power dissipation and maximizing performance?"

[1] Student Author

This paper proposes a mechanism in which the compiler analyses the assembly instruction stream to identify blocks of instructions, tag each block to one of the cores in the heterogeneous multi-core processor such that, if the block were to be executed in the corresponding core, it gives the maximum performance with minimum power dissipation. Once the code has been tagged and optimized, it then begins its execution in the first assigned core. When a change of tag is encountered, the thread is migrated to the core mentioned in the tag. Basically, this paper aims at shifting the book-keeping required for thread migration to the compiler so that the cost of hardware support is little.

The remainder of the paper is organized as follows. Section 2 describes the prior work done in this area. Section 3 describes the architectural assumptions made in the paper. Section 4 describes in detail, the algorithm used by our compiler. Section 5 discusses migration issues. Section 6 proposes a basic analytical model for power and performance. Section 7 shows the performance evaluations followed by the Conclusion.

2. RELATED WORKS

[6] proposes a general overview of heterogeneous multi-core processors, the benefits of migrating from a homogeneous to a heterogeneous core model, scheduling and software issues in a heterogeneous architecture. [8] gives a critical analysis of single ISA heterogeneous multi-core architectures. [5], a work by the same authors, describes the significant reduction in power dissipation as a result of judicious thread to core assignments. [7] proposes a dynamic mechanism of thread migration where execution traces and ratio of ISPs across two heterogeneous cores dictate thread migration. The need to dynamically monitor the ISP ratio across cores creates the

need for forced thread migrations, something of an overhead in the thread execution. [11] discusses a predictive approach for thread to core assignment in which similarities in program structure are exploited dynamically to perform scheduling. [10] discusses the issues involved in migration of a thread across cores. These issues encompass a critical analysis of all components right from cache hierarchy to register state.

With the above having mentioned, our work could be a considered as something which would complement the issues involved in thread migration. As the issues involved in thread migration with respect to the underlying hardware have been sufficiently discussed in [10], we restrict our discussion only to the methods that set the stage for and trigger off thread migration.

3. ARCHITECTURAL DESIGN

We have assumed a simple dual core single ISA heterogeneous architecture for our processor. The heterogeneity is purely at a theoretical level and we do not use a predefined set of attributes to explicate the heterogeneity. We term the processors as P (processor capable of extracting parallelism) and NP (processor incapable of extracting parallelism). A few features that may define the heterogeneity between P and NP are:

- Clock frequency.
- Functional unit distribution, number and function.
- Width of data/address paths.
- Pipeline structures.
- ILP extracting units like Scoreboards, Tomasulo reservation stations and Reorder buffers for speculative execution.

In addition to this, we assume the existence of ancillary hardware such as interconnects to support migration. More specifically, the paper assumes an ideal migration where the entire thread's state is transferred from one core to another without loss during the thread migration process. Although, we do take into account the time delay involved in the state transfer. We also require a change in the ISA of the heterogeneous core processor. We use an instruction called MIGRATE P (NP) which literally means “Migrate the thread to Processor P (NP)”.

4. CORE SPECIFIC BLOCK TAGGER (CSBT)

The CSBT is a unit of a compiler, the primary purpose of which is to tag blocks of assembly code to a core that would execute it with minimum power and without a compromise in performance. The CSBT should be aware of the underlying architectures of both the cores. The input to the CSBT is the assembly code that has been created after all compiler based optimizations. The output is an assembly code with the MIGRATE instructions inserted at vantage points. The algorithm used by the CSBT chiefly consists of the following 4 steps:

4.1 ILP based Block Splitting:

The available assembly code is scanned, disregarding the branches present in it and, blocks of statements are identified as either parallelizable or non-parallelizable. A block is placed under either of the above two classifications based on the following two factors:

- The instructions in a block do not have Data Dependencies over one another.
- The instructions in a block do not have Structural dependencies over one another.

Ideally, they must have a lot of ILP in them such that their execution time in a processor with ILP extracting units is significantly lower than in a simple processor, assuming both processors have the same state of execution. Once the code has been split into blocks, appropriate “MIGRATE” instructions are added at the end of each block.

4.2 Branch Target-based Migration:

Based on the target address of the branch instructions, appropriate “MIGRATE” instructions need to be inserted. This is based on the following three rules:

- If the “JUMP” instruction is in block labeled “X” (X=P or NP) and the target address goes to block labeled “X”, then no migrate instructions need to be inserted.

- If the "JUMP" instruction is in block labeled "X" and the target address is in the first half of a block labeled "Y", then the "MIGRATE" instruction at the top of block "Y" needs to be pulled down to the target address.
- If the "JUMP" instruction is in block labeled "X" and the target address is in the latter half of a block labeled "Y", then a "MIGRATE" instruction needs to be added at the target address.

4.3 Block Merging:

A threshold is set for the minimum size of a block. If such a small block is found, it is coalesced with one of its neighboring blocks. Further, if there are two blocks of the same type, they are coalesced. This step is mainly to eliminate blocks that are too small and to reduce the overhead caused due to too many migrations.

4.4 Re-addressing:

The address of the instructions, data and code references are changed after allocating fresh addresses to the program instruction due to the addition of the "MIGRATE" instructions.

5. MIGRATION ISSUES

Once the CSBT has tagged the assembly code with appropriate instructions, we can proceed with the execution of the code and the migration of the threads across cores. We examine the following issues involved in migration:

5.1 A Priori Migration:

Generally the migration overhead μ is a function of the amount of processor state to be transferred α, and the support of hardware for migration β.

$$\mu = f(\alpha, \beta) \qquad (1)$$

Since the CSBT tagged code already has the MIGRATE instruction inserted at appropriate locations, another pass on the CSBT can be added to insert AP MIGRATE (A priori MIGRATE) instructions sufficiently before MIGRATE instructions to perform a priori migration of processor state, which in turn, masks the migration overhead. This may, however require dirty bits to propagate post-migration state changes. How far the AP MIGRATE should be placed ahead of MIGRATE depends on the size of the state and the speed of migration interconnects.

5.2 Branch predictor type:

If the predictor is a correlating one, the amount of migration overhead increases with the amount of information, about the previous branches, to be transferred, which in turn increases with the degree of correlation of the predictor γ.

Now the Migration overhead is written as:

$$\mu = f(\alpha, \beta, \gamma) \qquad (2)$$

Hence if the predictor state also needs to be transferred, the positioning of the AP MIGRATE instruction should also consider this. There is, however, one nuance to be considered here. If the degree of correlation is very high, thus causing the AT MIGRATE instruction to be places much ahead of the MIGRATE instruction, there is a good chance of the occurrence of an intra-block branch between the AT MIGRATE and the MIGRATE instructions. If this were a single branch, very little information would be lost. However, if this were a tight loop, considerable branch predictor information is lost and this is reflected by the increased branch predictor warm-up time in the other core after migration. There is thus, a tradeoff between migration time and warm-up time.

5.3 Speculation:

If both the cores support speculative execution (which implies that the two cores come with good branch predictors), our mechanism can actually reduce the overhead incurred due to flushing the Re-Order buffers (ROBs) due to a misprediction. When a branch is encountered during the execution of a thread in Core 1, and if the branch is predicted to be taken, and if the target is a MIGRATE instruction, state is transferred to core 2 and speculative execution proceeds there. If the prediction were to be correct, the ROBs are committed. If the branch were mispredicted, then the execution can simply proceed on in core 1 and an interrupt can be sent to core 2 to clear the ROBs. If, however the next instruction in the sequence is a

MIGRATE, the migration can proceed in parallel to clearing the ROB. If the execution is speculated across branches and if the targets have migrations, then several migrations would occur and a single misprediction in a core X would lead to the clearing of ROB in the same core X which in turn would incur an overhead in performance. Thus, the amount of migration and flushing overhead is directly proportional to the degree of speculation δ.

$$\mu = f(\alpha, \beta, \gamma, \delta) \qquad (3)$$

6. MATHEMATICAL MODEL FOR PERFORMANCE:

Let x_1 be the average power consumed per instruction on the powerful processor P1, x_2 be the average power consumed per instruction on the ordinary processor P2.

$$x = x_1 - x_2 \qquad (4)$$

Let n_1 be the number of instructions executed that may be executed in parallel, n_2 be the number of instructions executed sequentially in a particular assembly program.

So the power consumed in the first processor is:

$$P1 = (n_1 + n_2) * x_1 \qquad (5)$$

The power consumed in the second processor is:

$$P2 = (n_1 + n_2) * x_2 \qquad (6)$$

In the scheme proposed, n_1 instructions are executed in P1 and the n_2 in P2.The power consumed in the proposed scheme is:

$$P = n_1 * x_1 + n_2 * x_2 \qquad (7)$$

The ratio P1:P is:

$$\frac{P1}{P} = \frac{(n_1 + n_2) * x_1}{n_1 * x_1 + n_2 * x_2} \qquad (8)$$

Since $x_1 > x_2$,

$$\frac{P1}{P} > 1 \qquad (9)$$

Further it may be noted that the performance doesn't vary. This is so because the n_2 instructions that can be executed sequentially do not have performance improvement in P1. This means they may be executed in the lower power consuming processor without any performance loss.

Now let us calculate the maximum number of blocks into which the CSBT can tag a code such that, instead of performing migration, the code can as well be executed in a single processor. Let k+1 represent that count. Hence k is the number of migrations. Assuming a constant migration overhead m and assuming the average time to execute an instruction in P_1 and P_2 as t_1 and t_2 ($t_1 < t_2$), we have (10) as:

$$\frac{1}{(n_1 * x_1 + n_2 * x_2)(n_1 * t_1 + n_2 * t_2 + km)} = \frac{1}{x_1 t_1 (n_1 + n_2)^2}$$

$$k = \frac{1}{m}\left((n_1 * t_1 + n_2 * t_2) - \frac{x_1 t_1 (n_1 + n_2)^2}{(n_1 * x_1 + n_2 * x_2)}\right) \qquad (11)$$

Similarly for the other processor P2, we replace $x_1 t_1$ by $x_2 t_2$. Knowing the parameters for the various processors and the constant hardware migration overhead m, k+1 can be calculated.

7. SIMULATION

We simulated our migration policy using GXemul 0.4.7.2 [5] and GCC MIPS cross compiler [4]. We used MIPS R3000 and R10000 processors for our simulation. The comparison between the two processors can be found in [2] and [3]. We simulated a software migration using shared memory. We used the CSBT to tag 20 programs with varying levels of dependencies but with uniform level of memory access. We normalized the block sizes and plotted the execution times as shown in Figure 1.

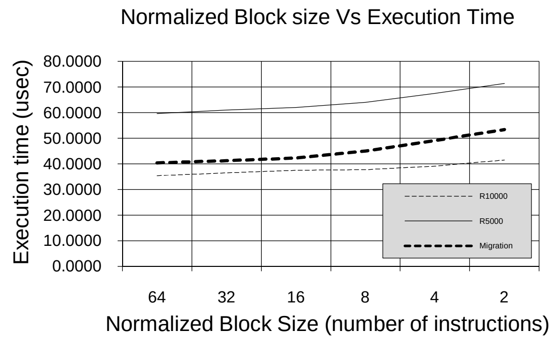

Figure 1. Execution time for codes with varying normalized block sizes.

As expected, complete thread execution in R10000 gave the best results for all block sizes. The migration results in a gradual increase in execution time because of the increasing migration overhead with the decrease in normalized block size (increase in the number of blocks). Now we plotted performance/power or 1/(Execution time*power). The result is shown in Figure 2. This gave better results for the migration over execution in R10000, particularly if the block sizes are large.

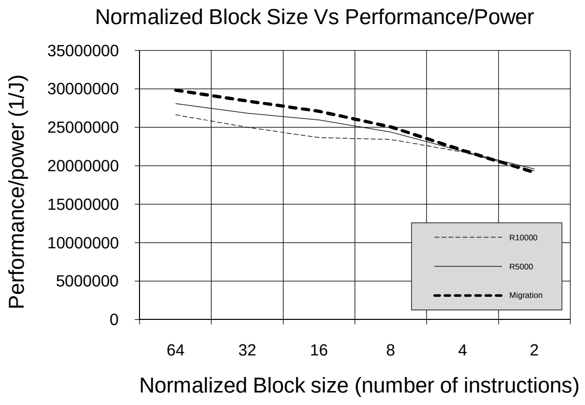

Figure 2. Performance/Power ratio for codes with varying normalized block sizes.

8. CONCLUSION AND FUTURE WORK

Thus we have seen how migration book-keeping can be shifted to the compiler just like shifting ILP extraction to the compiler. The CSBT can be added as an extra pass in any optimizing compiler to facilitate thread migration, principally to save power with no loss in performance. We have planned to implement our idea in m5 simulator by using single ISA heterogeneous dual-core chip consisting of Alpha EV5 and Alpha EV6 with necessary hardware support for migration and ISA support for MIGRATE and AP MIGRATE instructions and compare the results with IPC based dynamic migration. We have also planned to make a detailed analysis on the issues stated and examine how they affect overall performance.

9. REFERENCES

[1] http://en.wikipedia.org/wiki/R10000

[2] http://en.wikipedia.org/wiki/R5000

[3] http://gcc.gnu.org/

[4] http://gxemul.sourceforge.net/

[5] Kumar, R. Farkas, K. Jouppi, N.P. Ranganathan, P. Tullsen, D.M. Processor Power Reduction via Single-ISA Heterogeneous Multi-Core Architectures. Computer Architecture Letters. Volume: 2, Issue: 1. On page(s): 2- 2. January-December 2003.

[6] M. Gillespie. Preparing for the second stage of multicore hardware: Asymmetric (heterogeneous) cores. Technical report, Intel Corporation, July 2008.

[7] Michela Becchi , Patrick Crowley, Dynamic thread assignment on heterogeneous multiprocessor architectures, Proceedings of the 3rd conference on Computing frontiers, May 03-05, 2006, Ischia, Italy.

[8] Rakesh Kumar , Dean M. Tullsen , Parthasarathy Ranganathan , Norman P. Jouppi , Keith I. Farkas, Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance, Proceedings of the 31st annual international symposium on Computer architecture, p.64, June 19-23, 2004, MÃ¼nchen, Germany

[9] Saisanthosh Balakrishnan , Ravi Rajwar , Mike Upton , Konrad Lai, The Impact of Performance Asymmetry in Emerging Multicore Architectures, Proceedings of the 32nd annual international symposium on Computer Architecture, p.506-517, June 04-08, 2005

[10] Theofanis Constantinou , Yiannakis Sazeides , Pierre Michaud , Damien Fetis , Andre Seznec, Performance implications of single thread migration on a chip multi-core, ACM SIGARCH Computer Architecture News, v.33 n.4, November 2005

[11] Tyler Sondag , Viswanath Krishnamurthy , Hridesh Rajan, Predictive thread-to-core assignment on a heterogeneous multi-core processor, Proceedings of the 4th workshop on Programming languages and operating systems, October 18-18, 2007, Stevenson, Washington.