The HELIX Project: Overview and Directions

Simone Campanoni
Harvard University
Cambridge, USA
xan@eecs.harvard.edu

Timothy Jones
University of Cambridge
Cambridge, UK
timothy.jones@cl.cam.ac.uk

Glenn Holloway
Harvard University
Cambridge, USA
holloway@eecs.harvard.edu

Gu-Yeon Wei
Harvard University
Cambridge, USA
guyeon@eecs.harvard.edu

David Brooks
Harvard University
Cambridge, USA
dbrooks@eecs.harvard.edu

ABSTRACT

Parallelism has become the primary way to maximize processor performance and power efficiency. But because creating parallel programs by hand is difficult and prone to error, there is an urgent need for automatic ways of transforming conventional programs to exploit modern multicore systems. The HELIX compiler transformation is one such technique that has proven effective at parallelizing individual sequential programs automatically for a real six-core processor. We describe that transformation in the context of the broader HELIX research project, which aims to optimize the throughput of a multicore processor by coordinated changes in its architecture, its compiler, and its operating system. The goal is to make automatic parallelization mainstream in multiprogramming settings through *adaptive* algorithms for extracting and tuning thread-level parallelism.

Categories and Subject Descriptors

D.3.4 [**PROGRAMMING LANGUAGES**]: Processors—*Runtime environments*

General Terms

Performance, Languages

Keywords

Coarse grain parallelism extraction, runtime code adaptability, multiple programs

1. INTRODUCTION

By conventional definition, a "parallel program" is either expressed in terms of explicit parallel threads or tasks, or else is heavily annotated to guide compilers in mapping its data and control structures to parallel hardware. Research in recent years, however, has shown that in a very practical sense, every program is a parallel program, even one that has been designed and implemented with sequential semantics. Every long-running program depends on loops, and an increasing body of work demonstrates that automatic parallelization of loops, without help from the programmer, can lead to substantial speedup of the overall program [3, 6, 15, 10]. In fact, there have been discussions about whether parallelism should be explicit or not [2]. Since multicore microprocessors are now at the heart of devices from cell phones to supercomputers, it is important that these research demonstrations be translated soon into mainstream compilers and run-time software.

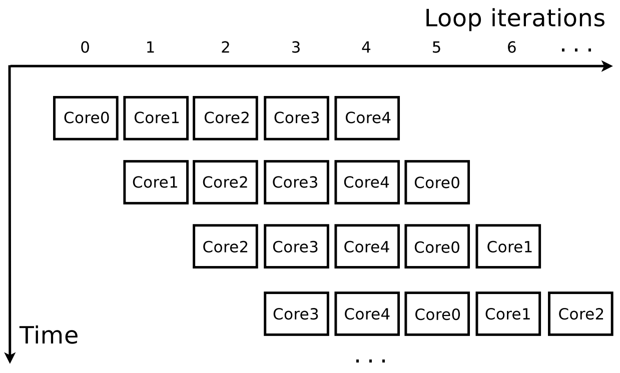

Figure 1: Cores execute loop iterations in round robin order.

That is the goal of our HELIX project. HELIX starts with a simple idea for loop transformation: to run a loop in parallel, assign its separate iterations to separate processing elements (cores) as shown in Figure 1. In general, the cores that handle separate iterations must communicate, both to synchronize and to exchange data. So successful parallelization of a loop depends on whether the benefit of running it in parallel outweighs the communication costs. When the separate iterations are independent, or nearly so, this simple approach to loop parallelization scales well with the number of available cores.

The reason this approach has not been more widely used is that historically the cost of communication between processing elements has swamped the benefits of running in parallel. Now that a powerful multiprocessor can come on a single chip, intercore communication costs are greatly reduced, and the trend is towards even greater reduction.

To show that the HELIX loop transformation is practical on a current commodity processor, we implemented a prototype compiler that parallelizes ordinary sequential code, including programs with irregular control and data behavior. The prototype

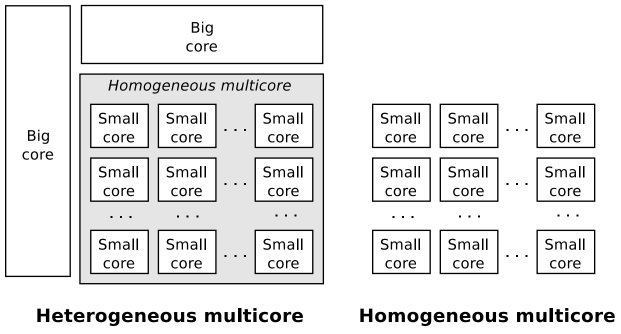

Figure 2: Classification of most multicore processor designs into heterogeneous and homogeneous solutions. In both cases, the majority of cores are homogeneous.

uses the memory system of the processor for communication between the hardware threads on separate cores that are executing separate loop iterations. To reduce latency, it couples each such iteration thread with a helper thread on the same core to force each inter-core signal to begin its journey (through shared cache) as soon as possible. One reason why the prototype is successful is that helper threads hide much of the cost of using the memory system for signaling.

Another reason that the HELIX prototype succeeds in producing significant overall speedups in workloads like the SPEC CPU2000 suite is that it is good at choosing which loops to parallelize and which to run in their original sequential form. It can select loops efficiently because the basic HELIX loop transformation is so simple. With the aid of a profile obtained as the program runs, HELIX can quickly estimate whether and by how much each loop will speed up if implemented in parallel. Since the speedup model accounts for the overhead of transferring program data between iteration threads, the loop selection heuristic tends to choose loops for parallelization that do not exchange much data between iterations.

Most multicore processor designs can be classified as either heterogeneous or homogeneous (see Figure 2). A homogeneous design is typically an array of relatively simple cores. Their number depends on the intended application (e.g., a sensor, a multimedia processor, an embedded system, or a commodity processor). A heterogeneous multicore processor generally augments such a homogeneous array with a small number of more complex cores, designed to run sequential code as fast as possible. HELIX is well suited to either the symmetric or asymmetric design. Parallelized loops run on the homogeneous array. If more powerful cores are also present, HELIX can use them for the parts of the program that run sequentially.

Our experience with the HELIX prototype is not intended to suggest that using the memory hierarchy and helper threads is the best way for parallel loop iterations to communicate. The experiment shows the benefit of reduced communication latency in a concrete way. Our speedup model is accurate enough to be used for predicting the effects of further overhead reduction. It shows that as ways are found to improve communication, the HELIX approach can achieve even better speedups.

One goal of the HELIX project is to investigate architectural enhancements that enable faster communication between cores. Another is to make HELIX-compiled programs more adaptable at run time. The HELIX prototype assumes that one program at a time uses the cores of its target processor, and the code of

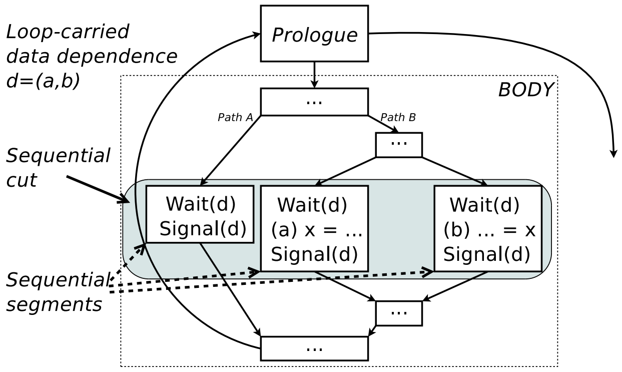
Loop-carried data dependence d=(a,b)
Prologue
BODY
...
Path A
Path B
Sequential cut
Wait(d) Signal(d)
Wait(d) (a) x = ... Signal(d)
Wait(d) (b) ... = x Signal(d)
Sequential segments

Figure 3: Code produced by the HELIX loop transformation. Depending on the execution path currently hot (path A or path B), the amount of parallelism within the loop iterations can change as the amount of code executed within sequential segments varies.

that program does not vary at run time. In reality, programs go through phases and their utilization of parallel resources can vary markedly from phase to phase. Furthermore, contention for such resources from other programs can change with time, as can the overall power target for the enclosing system. While HELIX will still use a static compiler to parallelize programs, a lightweight run-time thread will be added that can detect program phase changes and adjust the program's use of parallel cores accordingly. This lightweight run-time will also interact with the operating system, which will be modified to help schedule HELIX processes according to system load and its user's performance/power guidelines.

Section 2 describes the HELIX prototype in more detail, and Section 3 discusses the project's future directions. Section 4 locates HELIX with respect to related research, and Section 5 draws some conclusions from project so far.

2. HELIX PROTOTYPE

By constraining communication overhead, the HELIX loop transformation makes the simple idea of spawning different loop iterations on different cores efficient. The simplicity of the approach (and the resulting generated code) allows us to define a simple and accurate model for the speedup of a given loop. The transformation chooses the most profitable loops to parallelize automatically by using this speedup model, which relies on a profile obtained using representative input (e.g., the training input of SPEC benchmarks). Parallelized loops run one at a time. The iterations of each parallelized loop run in round-robin order on the cores of a single processor. The generated code can be adapted (even at run time) to use a different number of cores just by changing the mapping between loop iterations and cores.

The following paragraphs describe how HELIX minimizes inefficiencies that arise because certain code segments (known as *sequential segments*) must be executed in loop iteration order, and because of communication overhead, including both data transfer and synchronization. Our paper describing the HELIX prototype [6] implemented in the ILDJIT compilation framework [5] contains more detail.

Parallelism extracted.

Code not related to data dependences across loop boundaries

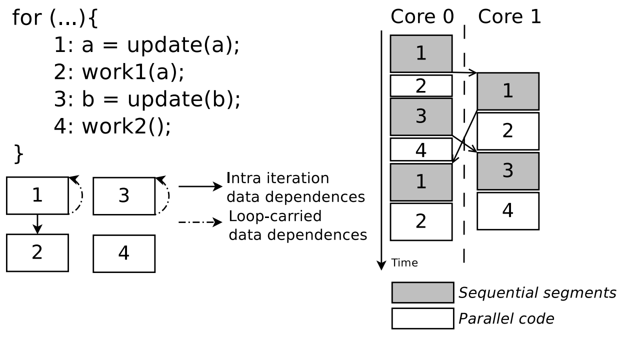

Figure 4: Execution of code produced by HELIX for a dual-core processor. Note that code blocks 1 and 3 must each be executed sequentially, but since they are independent, HELIX overlaps them in time.

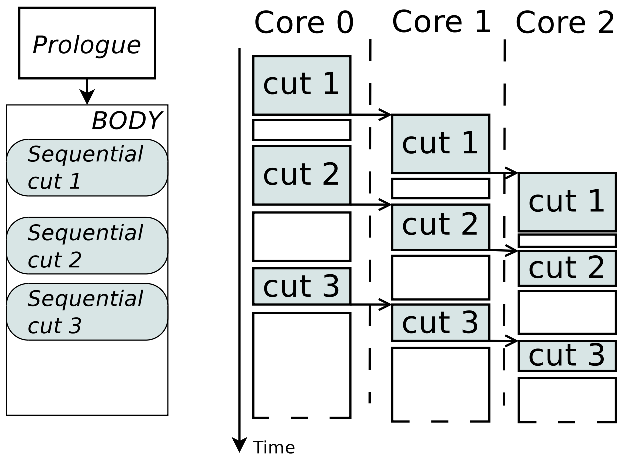

Figure 5: Sequential cuts created in the body of the loop due to loop-carried data dependences. The amount of parallelism among sequential cuts that HELIX is able to extract is shown on the right side.

is executed in parallel by different cores (code outside the sequential segments of Figure 3 and inside the white boxes of Figure 4). On the other hand, HELIX inserts code to ensure that the execution order of the remaining parts, the sequential segments, respects data dependences across loop boundaries, creating sequential cuts in the body (as shown in Figure 5) that must be traversed to end the execution of the body. The boundaries of these cuts are defined by data flow equations specifically designed for this purpose.

While the sequential segments of a given dependence must run in loop-iteration order, those of different dependences may run in parallel. HELIX executes distinct sequential segments concurrently whenever possible, as shown in Figure 4, where sequential segments 1 and 3 overlap.

Figure 6 shows the overall speedups achieved by HELIX. The geometric mean of the resulting speedups on a six core CPU is 2.25×, with a maximum of 4.12× (for art). Our experiments use an Intel® Core™ i7-980X with six cores, each operating at 3.33 GHz, with Turbo Boost disabled. The processor has three cache levels. The first two are private to each core and are 32KB and 256KB each. All cores share the last level 12MB cache, which is used to forward data values across cores of the same processor through the MESIF cache coherence protocol.

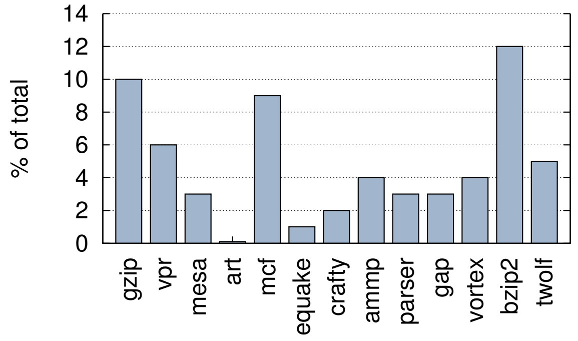

Figure 7: In the loops that HELIX chooses for parallelization, the fraction of potential data transfers that must be realized is small.

Communication overhead.

Execution overhead for the parallelized loops comes from two sources: data transfers and thread synchronizations.

Transferring data between threads to satisfy loop-carried data dependences is potentially a significant component of the overall overhead. However, as shown in [6] and summarized by Figure 7, in the loops that HELIX chooses for parallelization, the fraction of such potential transfers that must actually be realized is surprisingly small. In art, for example, only 0.1% of the data transfers are actually realized, which contributes to its large speedups in Figure 6.

Threads synchronize by sending signals. When a sequential segment ends, for example, it sends a signal to its successor thread to notify it that the corresponding sequential segment is free to start. HELIX minimizes the number of signals sent by exploiting redundancy among them. Moreover, as mentioned earlier, HELIX reduces the effective signal latency by exploiting the simultaneous multi-threading technology of the processor. It couples each thread that is running an iteration with a *helper thread* on the same core, whose role is to force the cache-mediated transmission of every inter-core signal to begin as soon the sending core makes it available.

3. THE FUTURE OF HELIX

The HELIX approach to parallelization aims to answer the following research questions: (i) How can microprocessors can be enhanced to speed inter-core communication? (ii) How can HELIX be extended to take advantage of faster inter-core communication? (iii) How can the parallel code produced by HELIX be used in a multiprogram scenario? (iv) How can HELIX tune the code at run time to accommodate the program's changing needs as it moves through different phases?

Changing the underlying hardware to further reduce inter-core communication overhead is the key to the success of our approach. We will extend HELIX's static compiler to take advantage of this faster inter-core communication. Moreover, we expect to extend the compiler to produce an additional lightweight run-time for each compiled program that makes it adaptable. This run-time collaborates with the OS to either acquire or re-

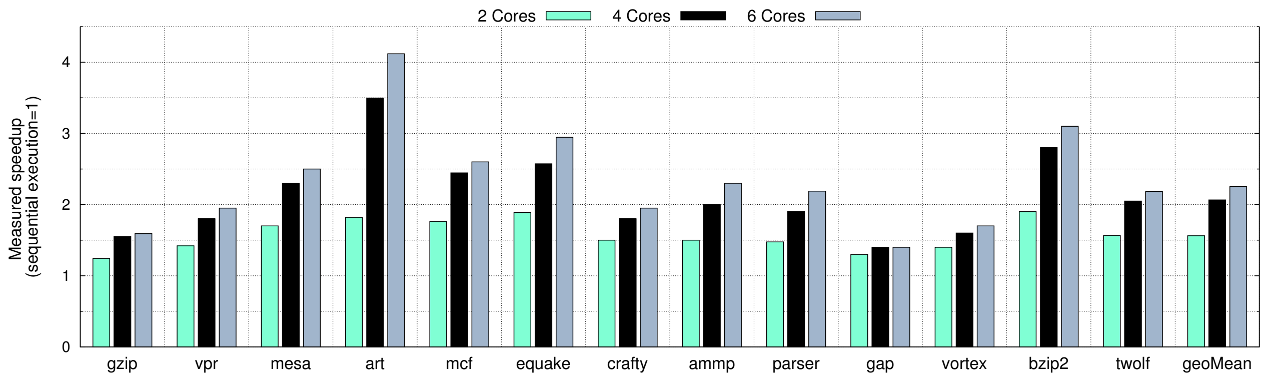

Figure 6: Speedups achieved by HELIX on a real system.

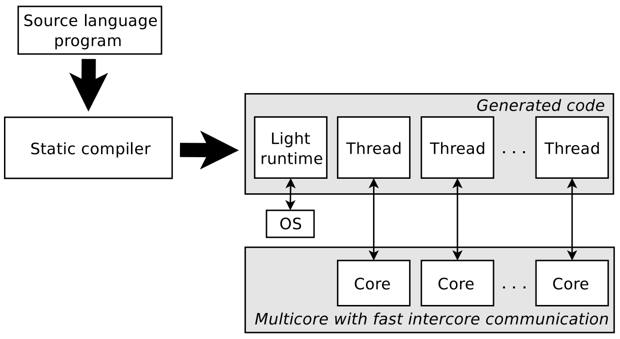

Figure 8: The main elements of the planned HELIX infrastructure and their principal interactions.

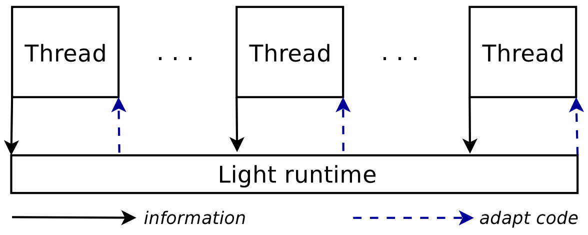

Figure 9: The run-time profiles the execution of the generated code to detect when statically defined rules match (solid lines). When they do, the run-time tunes the code by refining solutions created by the static compiler (dashed lines).

lease system resources. Through these negotiations the OS imposes constraints to balance the load of the overall system. Figure 8 shows the components just outlined.

3.1 Hardware Support

Currently, the HELIX prototype [6] targets commodity processors, which are not designed for frequent thread synchronizations. The key to increasing the number of cores that HELIX can accommodate is to reconsider the design of multicore processors. In particular, it is critical to find a small set of architectural changes that enable fast inter-core communication.

3.2 Static Compilation

The static compiler will extend the HELIX prototype [6] in two ways. First, if inter-core communication becomes faster, the profitability of parallelizing every loop changes, which makes a broader set of solutions available to the compiler. Second, the compiler will analyze the code to produce the most efficient runtime for achieving adaptability, as sketched in the next section.

3.3 Lightweight Run-Time and OS Support

Making parallelized code adaptable at run time depends on lightweight run-time support generated by the static compiler, together with help from the operating system. Code adaptability is important both for performance and for coexistence with other programs on the same system. This is because the resources an application requires and has access to can change at run time. Other researchers have also identified run time adaptation is important for achieving better overall parallelism [17]. Resource information is held by the operating system, but the compiler has the knowledge about how to best transform the program to make use of the available resources. Therefore, a run-time interaction is required which should be as unobtrusive as possible to avoid introducing overhead into the application.

Lightweight run-time.

It is well known that programs often go through different execution phases [19] that call for different optimizations. For example, the loop in Figure 3 might have a phase in which path A is taken exclusively followed by another phase where path B is taken exclusively. In the former phase, a larger fraction of the loop's execution time is spent running in parallel because path A includes less code that must execute in loop-iteration order. As this example shows, for HELIX, the best way of parallelizing a loop may depend on the phase. Moreover, since the profitability of loops can be different in different phases, loop selection is also affected.

To improve the performance of the running code, the run-time system applies a set of rules defined by the static compiler. The run-time system monitors the patterns of the rules. When it detects a match, it takes the corresponding action. A rule can be as simple as "if execution often leads to a given basic block, execute a certain loop in parallel". In this example, lightweight profiling needs to check the execution frequency of that basic block in case it becomes worth switching the loop to execute in parallel.

Figure 9 shows the use case of the lightweight run-time em-

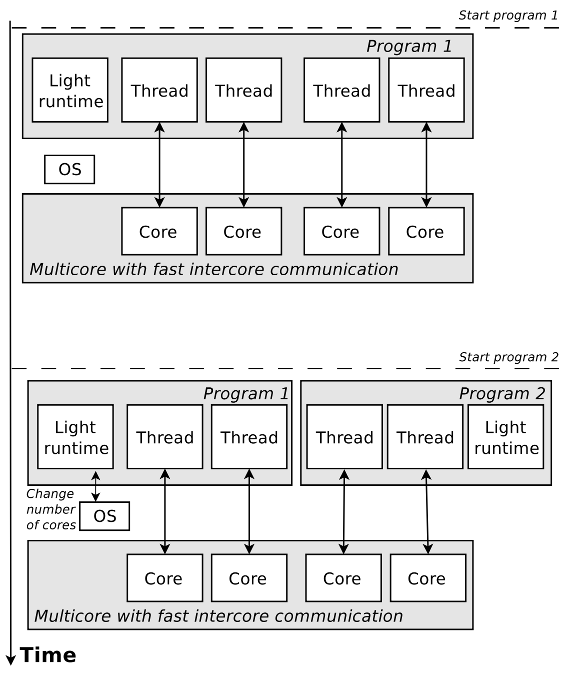

Figure 10: Example of code adaptation where a new program starts (i.e., *Program 2*) and it requires resources in use by a currently running process *Program 1*. In this example, the OS triggers a negotiation with the lightweight run-time to reduce the number of cores assigned to *Program 1* from 4 to 2.

bedded in the generated code when the objective is performance. This run-time can retrieve information such as execution paths taken, data communication patterns used, and thread synchronization patterns. If the run-time detects that is worth increasing the number of cores, it starts a negotiation with the OS.

The objective function of the run-time may not always be performance. For example, the OS may start to turn off parts of the chip if it detects that too much heat is being generated and thermal constraints are likely to be violated. In this situation, the run-time may be instructed to obtain the maximum performance for a given energy budget and will follow a different set of rules to achieve this.

OS support.

Figure 10 shows the use case for HELIX when the objective is coexistence between multiple programs. In this example, the lightweight run-time and the OS interact to adjust the number of cores in use based on the system load.

This interaction is a negotiation triggered by the OS when it decides to either reduce or increase the number of cores assigned to a given process based on the current load. The OS presents the lightweight run-time a hard and a soft threshold representing the maximum and the desired number of cores that the process can use, respectively. The choice of how many cores to use is left to the lightweight run-time, which must choose a number below the hard threshold and as close as possible to the soft threshold.

The lightweight run-time can also trigger negotiation when the needs of its program change. It can either request more cores when it comes to a highly parallel code section, or offer cores back to the OS when it detects a phase with limited parallelism.

In this interaction, each agent controls the resources that it knows best. The OS uses its complete view of the system to define the maximum and desired numbers of cores per process. The lightweight run-time uses the predictability of the HELIX loop transformation and its monitoring of the running code to determine the resource needs of its program.

4. RELATED WORK

There is a rich literature on parallelization of sequential programs by transforming loops into parallel threads of control. There are two main approaches: categorized into two principal paradigms: *pipelined multi-threading* and *cyclic multi-threading*.

Pipelined multi-threading (PMT).

Pipeline multi-threading techniques, the most established of which is called decoupled software pipelining (DSWP) [10, 15, 18], break loops into multiple threads such that cyclic data dependences never cross thread boundaries. The loose coupling of the resulting pipeline of threads allows data transfer between them to be buffered to prevent stalls in one thread from affecting others. The technique can produce significant speedups when this kind of parallelism is available in the program. Speculation has also been used to obtain speedups through the use of software transactional memory [16, 21].

The main drawback of PMT is that it restructures the code in a complex way that makes predicting the impact of this transformation code execution difficult. Therefore, it is unclear how to predict the speedup obtainable by applying these techniques to a given loop. That makes the problem of automatically choosing the most profitable loops for parallelization hard to solve. So for PMT, it is difficult to ensure that applying the transformation will not slow the program down.

Cyclic multi-threading (CMT).

Cyclic multi-threading techniques, such as DOALL and DOACROSS, target the parallelism between iterations of a given loop. The main drawback of these techniques comes from their high sensitivity to data communication overhead, which can easily lead to either slowdown or negligible speedup. The HELIX loop transformation belongs to this category, but it is able to constrain communication overhead enough to achieve significant speedups.

The closest approach to HELIX is the DOACROSS parallelization technique [8, 11], which has been studied in depth for regular workloads [1, 13, 20]. DOACROSS executes sequential segments without exploiting TLP between them [8]. Moreover, it does not permit either irregular control flow or irregular memory accesses within the loop [8]. Since HELIX has no such constraints and it considers a broader set of options during loop transformation, it can be seen as a generalization of the DOACROSS scheme that can be applied both to regular and irregular code.

Recent work on DOALL parallelism has used code transformations and thread-level speculation techniques to expose hidden parallelism in general purpose programs [22]. DSWP has also been mixed with DOALL [10, 18] to remove constraints on the number of threads extracted.

Run-time code adaptation.

Adapting code at run time in response to changes in program behavior has been studied deeply for managed code, such as Java or C#, in virtual machines [4]. However, these transformations can change the code quite significantly at run time. In contrast, the HELIX approach will be to fine tune code to adapt its execution, avoiding drastic transformations in order to minimize runtime overhead, which can be substantial for code parallelization techniques. There are also dynamic schemes to execute loop iterations in parallel at run time when they are detected to be independent [9, 23].

Helper threads.

Exploiting SMT to help critical threads was introduced in [7] and adapted to different domains later on [12, 14]. HELIX uses this scheme to solve the specific problem of fetching signals sent from another core.

5. CONCLUSION

The HELIX loop transformation shows that distributing loop iterations among cores of a real multicore processor can be effective even though it is not designed to support the necessary inter-core communication. The broader HELIX research project aims to design hardware more suitable for such transformations. It also adds corresponding enhancements to the static compiler and adds support to make the generated code adaptable to phase changes and availability of system resources. These include a lightweight run-time to adapt the program to the underlying system as its requirements change and allow coexistence with other applications. While HELIX has focused on sequential programs, there is no reason why it cannot work as well for explicitly multi-threaded programs.

Acknowledgements

This work was possible thanks to the sponsorship of Microsoft Research, HiPEAC, the Royal Academy of Engineering, EPSRC and the National Science Foundation (award number IIS-0926148). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

6. REFERENCES

[1] J. Allen and K. Kennedy. *Optimizing compilers for modern architectures*. Morgan Kaufmann, 2002.

[2] Arvind, David August, Keshav Pingali, Derek Chiou, Resit Sendag, and Joshua J. Yi. Programming multicores: Do applications programmers need to write explicitly parallel programs? *IEEE Micro*, 30, 2010.

[3] David I. August, Jialu Huang, Thomas B. Jablin, Hanjun Kim, Thomas R. Mason, Prakash Prabhu, Arun Raman, and Yun Zhang. Automatic extraction of parallelism from sequential code. In A. Tabatabai et al., editors, *Fundamentals of Multicore Software Development*. Chapman & Hall / CRC Press, 2011.

[4] Michael G. Burke, Jong-Deok Choi, Stephen J. Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, Vugranam C. Sreedhar, Harini Srinivasan, and John Whaley. The jalapeño dynamic optimizing compiler for java. In *Java Grande*, pages 129–141, 1999.

[5] Simone Campanoni, Giovanni Agosta, Stefano Crespi-Reghizzi, and Andrea Di Biagio. A highly flexible, parallel virtual machine: Design and experience of ILDJIT. *Softw. Pract. Exper.*, 2010.

[6] Simone Campanoni, Timothy M. Jones, Glenn Holloway, Vijay Janapa Reddi, Gu-Yeon Wei, and David Brooks. HELIX: Automatic Parallelization of Irregular Programs for Chip Multiprocessing. *CGO*, 2012.

[7] Robert S. Chappell, Jared Stark, Sangwook P. Kim, Steven K. Reinhardt, and Yale N. Patt. Simultaneous subordinate microthreading (SSMT). *ISCA*, 1999.

[8] R. Cytron. DOACROSS: Beyond vectorization for multiprocessors. *ICPP*, 1986.

[9] Francis Dang and Lawrence Rauchwerger. Speculative parallelization of partially parallel loops. In *5th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 285–299, 2000.

[10] Jialu Huang, Arun Raman, Thomas B. Jablin, Yun Zhang, Tzu-Han Hung, and David I. August. Decoupled software pipelining creates parallelization opportunities. *CGO*, 2010.

[11] Ali R. Hurson, Joford T. Lim, Krishna M. Kavi, and Ben Lee. Parallelization of DOALL and DOACROSS loops - a survey. *Advances in Computers*, 1997.

[12] Dongkeun Kim, Steve Shih wei Liao, Perry H. Wang, Juan del Cuvillo, Xinmin Tian, Xiang Zou, Hong Wang, Donald Yeung, Milind Girkar, and John P. Shen. Physical experimentation with prefetching helper threads on Intel's hyper-threaded processors. *CGO*, 2004.

[13] J. T. Lim, A. R. Hurson, K. Kavi, and B. Lee. A loop allocation policy for DOACROSS loops. *SPDP*, 1996.

[14] Chi-Keung Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. *SIGARCH Comp. Arch. News*, 2001.

[15] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. Automatic thread extraction with decoupled software pipelining. *MICRO*, 2005.

[16] Arun Raman, Hanjun Kim, Thomas R. Mason, Thomas B. Jablin, and David I. August. Speculative parallelization using software multi-threaded transactions. *ASPLOS*, 2010.

[17] Arun Raman, Hanjun Kim, Taewook Oh, Jae W. Lee, and David I. August. Parallelism orchestration using dope: the degree of parallelism executive. In *PLDI*, 2011.

[18] Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J. Bridges, and David I. August. Parallel-Stage decoupled software pipelining. *CGO*, 2008.

[19] Timothy Sherwood, Suleyman Sair, and Brad Calder. Phase tracking and prediction. In *ISCA*, pages 336–347, 2003.

[20] Hong-Men Su and Pen-Chung Yew. Efficient DOACROSS execution on distributed shared-memory multiprocessors. *ACM/IEEE conference on Supercomputing*, 1991.

[21] N. Vachharajani, Ram Rangan, Easwaran Raman, Matthew J. Bridges, Guilherme Ottoni, and David I. August. Speculative decoupled software pipelining. *PACT*, 2007.

[22] Hongtao Zhong, Mojtaba Mehrara, Steve Lieberman, and Scott Mahlke. Uncovering hidden loop level parallelism in sequential applications. *HPCA*, 2008.

[23] Xiaotong Zhuang, Alexandre E. Eichenberger, Yangchun Luo, Kevin O'Brien, and Kathryn M. O'Brien. Exploiting parallelism with dependence-aware scheduling. In *PACT*, pages 193–202, 2009.