2014 IEEE 26th Annual Software Technology Conference

Hybrid Hardware/Software Synergism

P. Burke, G. Albadarin, T. Maeshiro, P. Sweany
Dept. of Computer Science and Engineering
University of North Texas
Denton, Texas, USA
{PatrickBurke, GaithAlbadarin, TomyoMaeshiro}@my.unt.edu
sweany@cs.unt.edu

***Abstract*—Maximizing the efficiency of computing systems (hardware and software) in terms of power and processing is of paramount importance in the current age of revolutionary realtime hand-held computing devices. The packaging of heterogeneous computing devices (CPUs, ASICs, FPGAs, and associated communication protocols) on a single chip has been employed to mitigate issues such as power consumption and heat dissipation, but has yet to yield commensurate increases in computation efficiency or decreases in the cost of the software development for systems designed to operate on those devices. Existing compilers are amazingly efficient at code optimization and are capable of allowing some parallel processing by spreading the execution across a given number of available CPUs. The current off-the-shelf compiler technology is not, however, capable of spreading the processing across multiple non-homogeneous computing devices to provide the most efficient execution of the code for the given set of heterogeneous processing units. This is precisely the goal of our current Hybrid Computing (Hy-C) Compiler Project at the University of North Texas. Given a machine description file and source code, our Hy-C compiler is designed to partition the generated code to the specific types of computing devices that will provide the most efficient execution. Efficiency is measured in comparison to a set of constraints (power, throughput, heat, etc.) provided by the user. Thus, the goal of our compiler research will be to provide a novel capability of maximizing the execution potential of given software across a wide range of available processing components that can be combined on a single system on a chip.**

***Keyword* -- architecture description, retargetable compiler, partitioning.**

I. INTRODUCTION

We modern citizens of this increasingly interconnected world constantly demand newer, faster, and more powerful personal computing devices with a price well within the grasp of the average consumer. Furthermore, the commercial viability of these devices often depends on such non-technical parameters as size, color, and battery life. These high level "desirements" often conflict with each other forcing the designers to make informed compromises to provide the consumer with as much cutting-edge technology as is economically feasible. For example, state-of-the-art processor chips have a finite size that will, to some extent, dictate the minimum size of a cellular telephone. Likewise, the battery required to power that processor has a minimum size specification. It stands to reason that a larger battery can last longer, but the designers must limit the size of the battery to fit in a handheld consumer device. Of course, the phone could be used for days or weeks at a time without the need for recharging the battery if the battery could be the size of a car battery, but few, if any, consumers would find this a desirable solution. It is also worthy to note that processors, memory chips, and batteries all have heat-dissipation requirements that impose limits on the required dimensions of the physical device case. Once these physical trade-offs have been made and the final hardware has been selected, the software must then operate as efficiently as possible on the selected hardware platform.

A significant amount of software efficiency can be achieved with traditional compiler optimization techniques. Extracting parallel processing can also significantly improve the execution of software. Pipelining instructions provides micro-level parallel execution. Multi-tasking or multi-threading on a multicore processor, on the other hand, allows for full instruction-level parallelization. True parallel processing requires multiple processors simultaneously executing instructions that are data independent. While some compilers have shown some promise in automatically orchestrating this tasking, most have not. Thus, parallelization often becomes a task assigned to the system engineers and software architects to build into the development system itself. Any and all efficiency created in this manual mode will be at risk of being insidiously lost with each and every change in the software. Obviously, an automated compiler-driven solution is the preferred method of achieving a fully parallelized solution in software.

Our solution to this problem is a retargetable compiler named Hy-C (Hybrid Compiler). This compiler targets not only multiple processing units, but a heterogeneous set of processors that might include a mix of general purpose central processing units (CPU), field-programmable gate arrays (FPGA), digital signal processors (DSP), and Application-Specific Integrated Circuits (ASIP). These processors each have unique processing, energy

978-1-4799-5034-8/14 $31.00 © 2014 IEEE
DOI 10.1109/STC.2014.60

consumption, and thermal characteristics. The Hy-C compiler partitions the input source code to the computational devices that will provide the optimal performance in terms of processing throughput and energy consumption. In doing so, we achieve synergistic results which will yield better execution performance and lower energy consumption than the same code executed on a homogeneous processor set.

II. HISTORY

A discussion of the history behind this paper requires the discussion of two separate areas of study. The first, the history of the success of the software engineering industry, the track record, if you will, is necessary to determine the motivation that drives the need for a hybrid computing solution or any other improvements, for that matter. If current software development paradigms are producing sufficiently efficient and error-free software, it stands to reason that this rather drastic proposal would not be necessary. Perhaps, instead, it is just part of the continuing improvement process that follows all products.

The second area of study, processor history, addresses the question of why we cannot simply develop more efficient and powerful processors, just as we have since 1958 when Jack Kilby developed the first integrated circuit at Texas Instruments [1]. If the electronic industry can follow its time tested methods to develop new single processor chips that are more powerful and faster than ever before, why should the industry risk new technology like heterogeneous processors on a chip? If something is not broken, why fix it?

A. Software Engineering History

Despite 50+ years of practicing our trade, the software engineering industry is still plagued by failure after failure. A simple search for "software failures" will produce page after page of examples of truly epic system failures due to software errors. These are not *bugs*. The use of that term trivializes the grave nature of some of the errors. Errors are errors! Despite all of the existing software development tools and processes, errors are still occurring. The modern software systems are simply too complex for manual handling anymore. Automation is the only way to tame this beast.

Hardware engineers had the same problem in the 1960s with the advent of integrated circuits when hardware complexity grew beyond the reasonable grasp of the average or better engineer. That industry responded to the challenge with the introduction of Computer Aided Design (CAD) systems which simplified the task of creating increasingly complex hardware systems while "reducing errors and speeding design time"[2]. With the use of CAD systems, hardware engineers are now able to pack billions of transistors on a single chip. Software is still waiting for such simplification. Until that happens, increased fidelity of processes will have to suffice to control errors, but it is not working well. A few of the most notable cases of software failures are described below. Each of them had a very simple solution that should have been found to fix the errors long before any systems were fielded.

- *Year 2000*
  This was the error that never was an error. When I was taking my first undergraduate class in Computer Science in 1974, the instructor warned the students to ensure that all dates used in programming employed 4-digit year fields to avoid problems around the turn of the century. Twenty-six years later, the world waited for the impending doom in the first second of January 1, 2000. There was no doom. The sad part of this story is that despite billions of dollars spent on Y2K consultants throughout the world, the software industry failed in that it could not guarantee that nothing bad would happen at the stoke of midnight on December 31, 1999.
- *Mars Climate Orbiter (1999)* [3]
  In 1998 NASA launched the Mars Climate Orbiter which was due to arrive on station in orbit around Mars in late 1999. The mission of this spacecraft was to provide detailed information about the atmospheric temperature on Mars, dust, water vapor, and clouds. The mission was right on track through the enroute segment and handoff was made to the orbit team for orbit insertion. After sending the orbit insertion commands to the spacecraft, all contact was lost. The incident investigation team determined that the ground-based software provided by the contractor (Lockheed) produced an output in English units rather than the metric units that were specified in the contract. This mathematical error caused the spacecraft to crash into the surface of Mars. How could this obvious error have escaped all of the scrutiny that NASA employs in its world-class software process? How can this be prevented in the future?
- *Automatic brokerage problems on Wall Street (2012)* [4]
  On August 1, 2012, a glitch in newly installed automated brokerage software used by Knight Capital for automated trading caused a flurry of unintended trades. Before the staff could disable the software, it had lost more than $400 million in direct losses and a staggering loss of customers over the course of a few days. Of course, the Securities and Exchange Commission issued a stern statement of outrage about such unacceptable trading policies, but no details of the glitch were released to the public.

Just weeks ago (April 3, 2014) the Department of Defense announced that the F-35 Program would have to push initial operational testing out for at least 13 months due to software testing problems with the early releases of the software [5]. Software errors still make front page news.

All of these discussions provide strong evidence that the

current tools, practices, and processes employed by the software industry have fallen short of the goal of producing error free software products. It is our contention that the complexity inherent in those tools, practices, and processes have played a factor in their failure and we propose that simpler, not more complex, tools can help reverse this trend. The success of the hardware engineers using CAD tools certainly supports this theory. Thus, we propose the Hy-C compiler as a simple tool to increase the efficiency and correctness of software systems.

B. Processor History

"Moore's Law is a computing term that originated around 1970; the simplified version of this law states that processor speeds, or overall processing power for computers will double every two years" [6]. For the most part, this law has withstood the test of time. From the 1970's through the 1990's, advances in hardware design and materials allowed a steady increase in both clock speed and transistor density every year or two. While both increased density and increased clock speed result in higher thermal characteristics, simple fans and ambient airflow were sufficient to effectively dissipate heat. Furthermore, the thinner transistors required to manufacture processors of such high density caused significant increases in both power usage and heat generation. The result of this phenomenon was a decrease in processor performance increases. "Chip performance increased 60 percent per year in the 90s but slowed to 40 percent per year from 2000 to 2004, when performance increased by only 20 percent [7]."

The initial industry response to this situation was the introduction of chips with multiple cooler-running, more energy efficient processing cores instead of one increasingly powerful core. The individual processors in these chips do not necessarily run as fast as the high-end single processors, but they improve overall performance by handling more work in parallel. Unfortunately, the development of compilers that can efficiently extract the program execution parallelism required to take full advantage of the multicore solution have been slow to materialize. Thus, parallelization often becomes a manual task assigned to the development team to build the parallelism into the development system. As mentioned earlier, any and all efficiency created in this manual mode will be at risk of being insidiously lost with each and every change in the software.

III. Heterogeneous Processors On A Chip

The most recent entry in the processor field combines different types of processors, each with their own processing efficiencies and capabilities within a given niche, to allow efficient processing throughput in parallel with energy efficiency and a corresponding decrease in heat dissipation problems. These systems are not turn-key, however, in that the software tasking must somehow be divided in an efficient manner across this group of heterogeneous processors. As is the case described for multiprocessors above, this parallelization could be achieved through the manual design of a development system, but that is likely an expensive solution. Furthermore, changes in the software could be doubly expensive to ensure that no insidious errors are accidentally introduced. Once again, the obvious solution is to employ an automated compiler capable of balancing processing and energy efficiencies with simple input from the developers. Simplicity and automation are the hallmarks of our proposed Hy-C Retargetable Compiler.

IV. Hy-C Solution

The proposed Hy-C System is depicted in Fig. 1. Hy-C is a retargetable compiler that facilitates the distribution of executable code across a defined collection of heterogeneous processors taking full advantage of the processing/energy efficiencies of the constituent processors. Hy-C provides an efficient execution of the input source code on the fixed set of processors in accordance with the Objectives and Constraints. The processor set depicted in Fig. 1 is one possible collection of processor types that may be typical of a processor for use in hand-held personal devices. The System Specification contains an architectural description of the processing capability and the relative power-usage/thermal characteristics of each device including both processors and available memory. The other user-supplied file is the Objectives and Constraints which will prescribe the objectives for the run. One possible objective might be to run as fast as possible with no regard for power consumption. Alternatively, a user might want the system to run with minimal power consumption, regardless of processing speed. Realistically, users will probably specify something between these two extremes. The Partitioning process intelligently segments the source code into functional groupings based on the known processing capabilities of the available processors and directs those code segments to the applicable device for execution. If sent to the CPU device, the code is compiled by the native compiler for that processor. Partitioning uses graph matching to determine which code segments are sent to either the FPGA or ASIC. Of course, the functionality of an ASIC is well defined. For the initial work using FPGA, defined functions will be programmed into the FPGA. Ultimately, the FPGA gate routing code will be generated on the fly in later iterations. The processing time and power usage metrics are gathered in the Power Performance blocks and all of this data is sent to Optimization Control for performance analysis of the entire system. This function can be used to direct modifications to the partitioning design or, perhaps, to suggest a change in the hardware mix. Each of the Hy-C components will now be discussed in further detail.

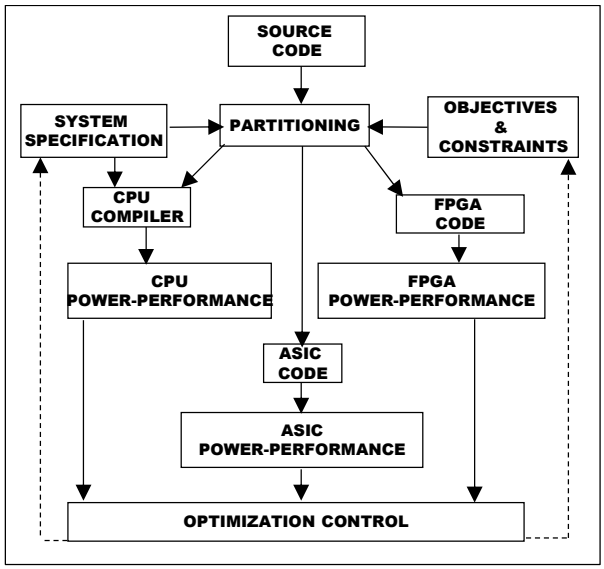

Figure 1. Hy-C Functional Diagram

A. System Specification

The System Specification contains the architectural description of the constituent processors and memory that allows the Hy-C system to be classified as a retargetable compiler. "A compiler for a fixed programming language is retargetable, if it can be adapted, so as to generate machine code for any processor within a defined class of processors, in such a way that the largest part of the compiler source code is retained."[8] In other words, a compiler is considered a retargetable compiler if the compiler itself does not have to be rewritten for each new target processor. This is not a new concept and, possibly to the surprise of most software engineers, it is not even uncommon. Most software engineers use a retargetable compiler on a daily basis, that is, if they use GNU GCC! Instructions for retargeting GCC can be found in the online documents provided by the GNU organization [9].

There are other notable retargetable technologies, one of which is LISATek, a complete retargetable compilation system developed within academia by Peter Marwedel and Ranier Leupers at RWTH Aachen University [10]. LISATek can generate all tools and models necessary for software development and for the integration and verification of embedded processing devices within a System on a Chip environment. It allows cycle-accurate modeling, even for pipelined architectures and produces both executable code and system simulators. As an added feature, it even produces VHDL and Verilog code that can be ported directly for the synthesis of Application Specific Integrate Circuits. After spawning numerous research papers, this product moved to the commercial world, but has not had widespread success in that arena.

One thing that GCC and LISATek do have in common, is that retargeting these systems for a new processor is far from simple. Each requires a significant investment in terms of cost, staff, and time to complete the retargeting. Looking at the architecture description requirements for these systems is a daunting task. They require in depth knowledge of general processor details and syntax to support the descriptions of the behaviors of each supported operation. This complexity and its associated cost prevent the routine retargeting of these systems. This is, perhaps, what has kept LISATek from becoming a booming technology in the electronics industry. Our approach with Hy-C is to keep the retargeting simple.

In order to support this simplistic approach, as previously stated, Hy-C uses only commercial processors that already have native compilers. Partitioning passes functional blocks of code to those compilers to produce the necessary machine code for execution. In a like manner, Partitioning passes functional blocks of code to the FPGA Coder to synthesize comparable code for those devices.

To support this version of retargeting, the Hy-C architecture description file will simply need to provide execution time, delay time, and energy consumption parameters for each computer operation, for each processor, and for each type of memory. The Partitioning function will use these data to determine the most efficient routing for each functional block of code as defined by the System Objectives and Constraints. These constraints could, for example, favor minimal execution time over low power consumption. On the other hand, the user may specify that low power consumption is the desired outcome without regard for optimal processing time performance. Realistically, the constraints will probably specify some sort of tradeoff between these performance parameters. While determining the time and energy parameters will not be trivial, it is not anticipated that it will be nearly as difficult as generically describing the full processing behaviors of each instruction.

The other task of Partitioning is to provide synchronization code to the Program Controller to ensure the proper sequencing of the code, start to finish, in accordance with the intended sequencing of the original source code. The result of this partitioning will be the achievement of the maximum amount of parallel execution for the input source code.

B. Partitioning

A compiler that partitions computation among a hybrid chip's heterogeneous processing elements provides a major component of our code-generation strategy. As a general problem, partitioning tasks among multiple dependent resources is an NP-complete problem [11]. However, researchers have successfully employed many heuristics to obtain good solutions, in reasonable time, for specific instances of the problem. In our case, we must partition tasks among heterogeneous resources on-chip. This

requirement, along with other constraints, makes the partitioning compiler difficult to design. However, in addition to using algorithms and techniques employed in hardware-software co-design, we use the design, algorithms, and code of existing compilers, namely Rocket [12] and LLVM [13].

The hardware/software co-design community has a long history of partitioning computation among heterogeneous resources. However, we concur with Brandolese et al. [14] that most suffer from one or both of two problems. Either they focus on too narrow an application area or they require too much ``pre-design'' by domain experts, which often leads to the same problem, i.e. too narrow a focus for a wide range of applications. In fact, much of the hardware/software literature describes different heuristics used to find acceptable solutions in a vast search space. We have used similar approaches to finding acceptable solutions to NP-Complete problems such as instruction scheduling and register partitioning relying on genetic algorithms [15][16][17]) and simulated annealing [18].

However, for this particular partitioning problem we have had good success with an algorithm that builds data dependence graphs (DDGs) [19] for each function of the C code and determines which available computing resource best fits the parallelism available in that DDG. We define the parallelism in this case as the number of nodes in the DDG divided by the time-sensitive critical path of the DDG.

C. CPU Compiler

A hallmark of Hy-C is that it has been kept as simple as possible for the user. By utilizing off-the-shelf processors for which commercial compilers are readily available, Hy-C only requires that the user provide an estimate of the processing time and heat generation characteristics for use in extracting execution time and power-usage efficiency in the compilation process. Other retargetable compilers, GCC and LISATek, for example, require large staffs of knowledgeable engineers along with a large budget and ample schedule time to choreograph the retargeting of a new processor. This is not the efficiency model required by successful telecom companies in this fast-changing industry!

In terms of speed and thermal efficiency, the CPU is assumed to be the slowest processor and it will probably produce the greatest heat for a given execution of source code.

D. FPGA Code

Generating on-the-fly, application specific FPGA code is certainly a hard problem, but there are some available commercial solutions for converting code written in C to VHDL for execution in FPGAs which could simplify this problem. In the short term, however, the initial version of Hy-C will employ the use of some pre-determined functions whose execution on FPGAs are known to be efficient. These functions will be hand-coded using VHDL, Verilog, or C (along with a VHDL/Verilog converter) to produce the FPGA "code". Standard compiler techniques will be used to build a representative graph (DDG) describing the code sequence that can be assigned to the FPGA. The Partitioning function will then use subgraph matching techniques to choose source code that will be directed to the FPGA for execution. Later versions of Hy-C will tackle the customer-generated FPGA code solution.

At this point we see the main impact of FPGA code as allowing more parallelism than is inherent in any fixed CPU processor. As an example, consider matrix multiply. Since the innermost loop contains no loop-carried dependence, the only limit to parallelism is the number of multiply-accumulate functions that can be started at any point in time.

Consider TI's family of DSP chips. They allow up to eight distinct operations to complete in one cycle. In theory, then, these chips allow for reading four source operands (for two loop iteration's multiply-accumulates), two multiply-accumulates for different loop iterations, and for two writes of values for yet two more loop iterations, completing two inner-loop iterations per cycle. But, could we do better? For a fixed CPU, we could only do better if more functional units are available. In contrast, consider using a hybrid architecture's FPGA to perform multiply-accumulates in parallel. Instead of being limited by a fixed processor's hardware, we may compute as many inner loop iterations in parallel as we can program in the FPGA. So the question of interest becomes: With the available FPGA resources, how many parallel sets of two loads (for input values), one multiply-accumulate, and one store can be executed in parallel?

To answer this, we compared a matrix multiply in which the innermost loop was pipelined in FPGA to one executed in a CPU. We assume all data are prefetched into FPGA (Xilinx Vertex5 XC5VLX220, the most powerful FPGA at the time we did this short study). The process to compute matrix c[i][j] was parallelized and pipelined so that it could be completed in one clock cycle. The results are summarized in Table I.

We limited our calculations at N=32 since for a size of 64 would exhaust the resources of the FPGA we were using at the time. Still, we can see that using the FPGA we can improve inner loop performance by a factor of 25. And with significant improvement in size, speed and programmability of FPGAs recently, the improvement would be much greater.

E. ASIC Code

An ASIC is shown in Fig. 1, but the initial Hy-C will not actually support ASICs. In general, FPGAs and ASICs have comparable performance characteristics. The choice of one over the other is largely a matter of development cost and

TABLE I. MATRIX MULTIPLY COMPARISON

	N=2	N=4	N=8	N=16	N=32
CPU (ns)	110	875	5,500	42,500	350,000
FPGA (ns)=n² x clock = FPGA (ns)	28	109	635	2,549	13,865

time. Unlike FPGAs which can be reprogrammed time and time again, ASICs are manufactured with the gate logic hard coded in the design of the device. In other words, they are not programmable at all after manufacture. There is an expensive non-recurring engineering cost associated with ASICs to manufacture the proper logic into the logic gates. This takes both time and a significant amount of cost. The benefit, however, is that the manufacture of the device us considerably cheaper than that of a FPGA. As such, the non-recurring cost of the up front engineering can be negated by the savings in manufacturing if a sufficient quantity of devices are needed to affect this saving. On the other hand, once the device is manufactured, there can be no modification without another investment in the development cost of a new ASIC. Therefore, system designers must consider all of these factors in determining whether or not to include ASICs in the final heterogeneous processor.

F. Power/Performance Modules

The three Power/Performance modules are depicted in Fig. 1, one for each constituent processor. The purpose of these modules is to gather performance metrics for each of the respective devices for analysis by the Optimization Control. The metrics collected must include such things as the execution time of code on the device and the power consumed by the device in the execution of the assigned software. Other metrics may be required to satisfy specific customer needs as described in the Objectives and Constraints.

G. Optimization Control

The Optimization Control must evaluate the collective performance data produced by Hy-C from the processing of a run of the user-supplied source code. The data generated by the Optimization Control function can then be used to modify the Partitioning algorithms or, possibly, to provide evidence that a different hardware configuration may be more efficient. The data might also suggest that the user change the Objectives and Constraints. If, for example, the run indicates that one device is using an excessive amount of power, thus generating excessive heat, the Partitioning algorithm might have to change to favor reduced power consumption over processing time performance.

V. FUTURE WORK

The current vision for Hy-C works only with a static hardware configuration. All decisions relative to the modification of the mix of processors is a manual operation which is guided by the performance analysis data provided by Hy-C. A logical improvement to the Hy-C system would be to automate this function to suggest possible hardware changes that might provide more efficient execution of the test software in terms of processing speed, power consumption, or heat generation.

VI. CONCLUSIION

We anticipate that the Hardware/Software Co-Design process will benefit from the Hy-C Retargetable Compiler by providing efficient code execution results (a good trade-off of processing time, heat generation, and power consumption for a given configuration of hardware components). The Hy-C solution will yield execution times as fast, or faster than, any of the constituent processors operated in isolation. As an added benefit, it will provide the user with power consumption estimates for that solution, which may assist in the evaluation of the constituent hardware mix. Using the native compilers for constituent off-the-shelf CPUs makes Hy-C very user-friendly by eliminating the need for time consuming and complicated architecture descriptions, which has been the bane of the retargetable compiler industry since its inception. The simplicity of Hy-C for the user combined with its powerful partitioning and processing capability should be a step in the right direction for reducing future software errors.

REFERENCES

[1] "Integrated Circuit Invented by Jack Kilby," Texas Instruments Website, http://www.ti.com/corp/docs/company/history/timeline/semicon/1950/docs/58ic_kilby.htm (accessed 3 April 2014).

[2] "1966 - Computer Aided Design Tools Developed for IC's," http://www.computerhistory.org/semiconductor/timeline/1966-CAD.html (accessed 4 April 2014).

[3] "Likely cause of orbiter loss found," [Online] http://mars.jpl.nasa.gov/msp98/orbiter/. 1999 (accessed 4 April 2014).

[4] "Automated trading: The Lazarus of Wall Street," [Online] http://www.economist.com/node/21560306. (accessed 4 April 2014).

[5] [Online] http://www.foxnews.com/tech/2014/04/03/f-35-fighters-plagued-with-delays-cost-overruns-federal-report-says/ (accessed 4 April 2014).

[6] "Moore's Law or how overall processing power for computers will double every two years," [Online] http://www.mooreslaw.org (accessed 4 April 2014).

[7] D. Geer. "Chip makers turn to multicore processors," *IEEE Computer*, May 2005, pp. 11-13.

[8] R. Leupers. *Retargetable Code Generation for Digital Signal Processors*, Kluwer Academic Publishers, Boston, 1997, p. 21.

[9] GCC online documents, [Online] http://gcc.gnu.org/onlinedocs/ (accessed 4 April 2014).

[10] R. Leupers and P. Marwedel, *Retargetable Compiler Technology for Embedded Systems: Tools and Applications*, Kluwer Academic Publishers, Boston, 2010, p.134.

[11] M. Gary and D. Johnson, *Computers and intractability: A guide to the theory of NP-completeness*, San Francisco, CA: W.H. Freeman and Company, 1979.

[12] P. Sweany and S. Beaty, "Overview of the Rocket retargetable C compiler," Dept. of Comp. Sci., Michigan Tech. Univ., Houghton, Tech. Rep. CS-94-01, Jan., 1994.

[13] C. Lattner, "LLVM: An infrastructure for multi-stage optimization, Masters thesis, Comp. Sci. Dept., Univ. of Ill. at Urbana-Champaign, Urbana, IL, 2002 (see http://llvm.cs.uiuc.edu).

[14] C. Brandolese, W. Fornaciari, L. Pomante, F. Salise, D. Sciuto. "Affinity-Driven Design Exploration for Heterogeneous

Multiprocessor SoC," *IEEE Transactions on Computers* 55(5): 508-519, 2006.
[15] S.J. Beaty, "Instruction scheduling using genetic algorithms," Ph.D dissertation, Mech. Eng. Dept., Colorado St. Univ., Fort Collins, CO, 1991.
[16] S. Beaty, S. Colcord, and P. Sweany, "Using genetic algorithms to fine-tune instruction scheduling," in *Proc. of the MPCS*, 1996.
[17] D. Sule, "Evaluating register bank partitioning with genetic algorithms," in *Proc. of the 7th Int. Conf. on MPCS, 2002.*
[18] P. Sweany and S. Beaty, "Instruction scheduling using simulated annealing," in *Proc. of the MPCS*, 1998.
[19] V. Allan, S. Beaty, B. Su, and P. Sweany, "Building a retargetable local instruction scheduler," *Software Practice and Experience*, vol. 28, no. 3, pp. 249-284, Mar, 1998.