A Framework For Compiler Driven Design Space Exploration For Embedded System Customization*

Krishna V. Palem, Lakshmi N. Chakrapani, Sudhakar Yalamanchili

Center for Research on Embedded Systems and Technology,
Georgia Institute of Technology, Atlanta GA 30308, USA.
(palem,nsimhan,sudha)@ece.gatech.edu
(http://www.crest.gatech.edu)

Abstract. Designing custom solutions has been central to meeting a range of stringent and specialized needs of embedded computing, along such dimensions as physical size, power consumption, and performance that includes real-time behavior. For this trend to continue, we must find ways to overcome the twin hurdles of rising non-recurring engineering (NRE) costs and decreasing time-to-market windows by providing major improvements in designer productivity. This paper presents compiler directed design space exploration as a framework for articulating, formulating, and implementing global optimizations for embedded systems customization, where the design space is spanned by parametric representations of both candidate compiler optimizations and architecture parameters, and the navigation of the design space is driven by quantifiable, machine independent metrics. This paper describes the elements of such a framework and an example of its application.

1 Introduction

Over the last two decades Moore's Law has remained the primary semiconductor market driver for the emergence and proliferation of embedded systems. Chip densities have being doubling and cost halving every 18 months. For this trend to continue, the non-recurring engineering (NRE) costs must be amortized over high unit volumes, thus making customization attractive only in the context of high-volume applications. This generally implies uniform designs for a large set of applications. However, customization is central to a range of existing and emerging embedded applications including high-performance networking, consumer and medical electronics, industrial automation and control, electronic textiles and computer security, to name a few. These applications are characterized by evolving computational requirements in moderate volumes with stringent time-to-market pressures.

In the past, customization meant investing the time and money to design an application specific integrated circuit (ASIC). However, the non-recurring engineering (NRE)

* This work was supported in part by DARPA PCA program under contract #F33615-03-C-4105 and DARPA DIS program under contract #F33165-99-1-1499. A preliminary version of this paper appeared in the Proceedings of the Ninth Asian Computing Science Conference, December 2004

cost of ASIC product development has been rapidly escalating with each new generation of device technology. For example, a leading edge embedded system based on an 80 million transistor custom chip in 100 nm technology was projected to cost upwards of $80M [13]. Less aggressive chip designs still incur significant NRE costs in the range of tens of millions of dollars with mask costs alone having surpassed $1M. Furthermore, product life cycles and competitive pressures are placing increasing demands on shortening the time-to-market. This in turn has been a significant limiting factor to the growth of the industry that is increasingly demanding customized solutions. If the embedded systems industry is to grow at the projected rates it is essential that the twin hurdles of increasing NRE costs and longer time-to-market be overcome [24].

Thus, to sustain Moore's Law and its corollary through the next decade, a major challenge for embedded systems design is to produce several orders of magnitude improvement in designer productivity to concurrently reduce both NRE costs and time-to-market. The required gains in productivity and cost can only be realized by fundamental shifts in design strategy rather than evolutionary techniques that retain legacy barriers between hardware and software design me thodologies. This paper presents an overview of compiler directed design space exploration as a framework for addressing the challenges of embedded systems customization for the next decade.

2 Evolution of Approaches to Customization

Increasing hardware design complexity and design costs have historically driven innovations in the development of higher levels of hardware abstractions and accompanying design tools for translation into efficient hardware designs. The concept of automated, customized hardware design, termed *silicon compilation*, referred to automated approaches for translating high level specifications to hardware implementations that were subsequently subjected to hardware-centric optimizations. A departure from this conventional view articulated a new approach wherein optimizations were applied early in the compilation process to more abstract representations of the hardware [19]. This framework enabled the development of computationally tractable solutions to several geometric optimizations that, at the time, were applied to lower level hardware descriptions. As the progression of Moore's Law continued through the next decade, each new technology generation placed formidable challenges to designer productivity. Silicon compilation consequently evolved into two distinct disciplines: optimizing compilers and electronic design automation (EDA).

Through the 1970's instruction set architectures (ISAs) emerged as a vehicle for focusing the NRE costs of processor hardware development. Through re-use via software customization, the processor hardware NRE cost could be amortized through high volumes achieved by using the processor across many applications. As technology moved through the sub-micron barrier, to sustain the cost and performance benefits of Moore's Law, we saw the emergence of reduced instruction set architectures (RISC) and accompanying optimizing compiler technology in the early 80's. For the next 15 years the demands for instruction level parallel (ILP) processors and automated parallelization drove developments in compiler technology in the areas of program analysis, e.g., array dependence analysis, and program transformations eg. loop transformations. The goal

of program analysis and transformation was to optimize the execution of a program on a fixed hardware target that was abstracted in the form of an ISA. Program transformations also evolved to exploit facets of the micro-architecture and memory hierarchy that were exposed to the compiler, such as the cache block size, memory latency and functional unit latencies. However, in all of these cases the target hardware remained fixed. Central to these program analysis and transformation techniques was the evolution of rich graph-based intermediate program representations, or *program graphs*.

Independently, and concurrently, electronic design automation (EDA) continued to innovate in automating the design of increasingly complex generations of chips. The abstractions for automated EDA tool flows evolved from switch-level abstractions of transistors, through the gate-level, to the register transfer level, with much of the current interest focused at the Electronic System Level (ESL), utilizing hardware specifications in the form of languages such as SystemC and System Verilog. The ISA remained the primary interface between program graphs (software) and the hardware. Decisions concerning functionality that should be in hardware, versus that which should be implemented in software were made with respect to subsequently inflexible and rigid partitions between software and hardware implementations. Thus, through the 80's and much of the 90's the disciplines of optimizing compiler technology for microprocessors and EDA technology for the development of ASICs matured into distinct, vertically integrated design flows with little to no cross-fertilization.

In the 90's a few research efforts began investigating design techniques for customization of embedded systems wherein the compiler optimizations were applied in the context of traditional EDA design tasks. The Program-In-Chip-Out (PICO) system at HP Laboratories [28] pioneered an approach that leverages well known program transformations such as software pipelining to synthesize non-programmable hardware accelerators to implement loop nests found in ANSI C programs [29]. The PICO system also synthesizes a companion custom Very Long Instruction Word (VLIW) processor that is selected through a framework for trading area (through the number of functional units) for performance (through compiler scheduling of instructions) [28]. While PICO generates custom ASIC solutions, the Adaptive Explicitly Parallel Instruction Computing (AEPIC) model [22, 17] targets an EPIC processor coupled to a reconfigurable fabric such as a field programmable gate array (FPGA). Custom instructions are identified from an analysis of the program while hardware implementation of these custom instructions are realized in the reconfigurable fabric.

The EPIC ISA is extended with instructions to communicate with the custom fabric. Traditional compiler optimizations for register allocation, instruction scheduling, and latency hiding are extended to manage the reconfigurable fabric. While hardware description languages emerged as the dominant specification vehicle of choice for hardware design, Handel-C emerged as a C-based language for specifying hardware attributes of concurrency and timing [1] for hardware synthesis leading to a C-based design flow for hardware/software partitioning and hardware synthesis. More recent versions of C-based design flows include ImpulseC [2], SilverC [4], and Single-Assignment C [27]. While all of these approaches attempt to leverage the C language and use an HDL as an intermediate representation, the dynamically variable instruction set architecture approach [11] [18] compiled ANSI C programs to FPGA fabrics without

a HDL intermediate representation. In the same spirit of using programming language based specification [25], a more recent effort describes a comprehensive approach to customization for embedded control applications [7]. A high level specification based on an extended finite state machine formalism is analyzed and used for partitioning functionality into hardware and software driven by performance constraints. The target architecture is an embedded processor coupled with a reconfigurable fabric. Another recent approach for the customization of the memory hierarchy combines locality enhancing transformations with exploration of cache design parameters for programs with pointers – ubiquitous to embedded applications (e.g., C) – and aimed at minimizing the physical size and energy consumed by embedded cache memories [21] [20]. The results demonstrated halving of the cache size and energy consumption, while in many cases simultaneously improving the performance by a similar factor. Thus, a powerful algorithm based on the principles of average-case analysis could achieve power and cost (silicon area) improvements corresponding to those achieved by an entire generation of Moore's Law. Several other recent projects have also begun exploring the co-operative use of optimizing compilers and behavioral synthesis compilers recognizing the strengths of each and seeking a principled application of both targeted to FPGAs [31] [15] [14]. These research efforts further underscore the trend towards the cross-fertilization between the disciplines of optimizing compilers and EDA.

The productivity and performance improvements required to sustain Moore's Law through the next decade must come from a major shift in design principles rather than continued evolutionary improvements in separate, vertically integrated design flows for optimizing compiler technology and EDA. Global optimizations that span these two disciplines afford opportunities for major improvement in designer productivity and system performance. Central to the development of such analysis and optimization techniques will be the redefinition of the hardware/software interface, to enable customization techniques that can operate directly on program graph representations, in bridging to hardware abstractions and thus avoiding the performance and productivity hurdles of utilizing a fixed ISA customized for an application. Design space exploration is a framework for articulating, designing, and implementing such global optimizations where the design space is spanned by parametric representations of both candidate compiler optimizations and architecture parameters and the navigation of the design space is driven by quantifiable, machine independent metrics. In a manner reminiscent of the re-definition of the hardware-software interface that took place in innovating RISC architectures, productivity improvements will come from re-defining the interface between, and roles of, optimizing compilers and EDA tools.

3 Compiler-Directed Design Space Exploration

Functionally, the process of customization via design space exploration (DSE) transforms both a program and the target architecture to generate a ”best fit”. The essential elements of DSE are describe in this section.

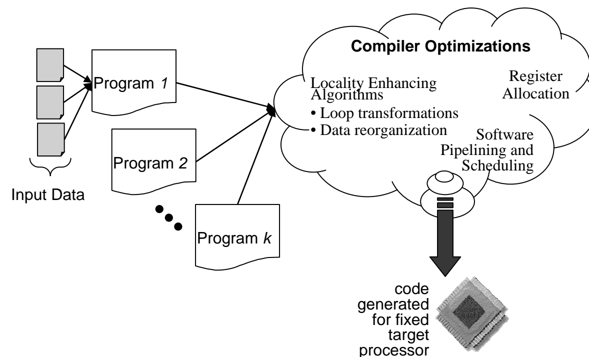

Fig. 1. The traditional role of a compiler: Compiling to a fixed target

3.1 Compiler As a Vehicle For Design Space Exploration

As illustrated in figure 1, traditionally compilers were designed to target a fixed architecture with a rigid hardware-software interface in the form of an ISA. Compiler optimizations were developed with the aim of efficient utilization of fixed architectural resources such as storage (eg. registers) and data-path (eg. functional units). However, since the hardware is fixed at design time, hardware vendors are motivated to produce designs that appeal to the largest base of applications so as to amortize the NRE costs. Consequently, performance for a specific or small subset of key applications kernels is sacrificed. What is desirable is that the application and architecture be *co-optimized* and the design space of compiler optimizations and architectural alternatives be *co-explored*.

The application of compiler directed DSE is illustrated in Figure 2 (a). In this example, aspects of the architecture are configurable and *under the control of the compiler.* Optimizations now target a space of alternative customized micro-architectures seeking one which minimizes or maximizes an objective function. As an example, consider a data layout optimization for large arrays of data. Rather than seeking a layout that optimizes a fixed cache block size, the optimization can concurrently search a space of block sizes and data layouts that would maximize performance. For the first time, a compiler views *an application as a mutable entity which is co-optimized with the target architecture for a better "fit" and better designs*. By contrast, other hardware-software codesign methodologies have largely viewed the application as an immutable or "fixed" entity for which the target architecture is optimized. Our framework, as illustrated in figure 2 is a technique for integrated compiler/EDA optimization, breaking the traditional hardware-software barrier and incorporating parametric descriptions of compiler optimizations as well as the target architecture into the design space. Figure 2 (b) illustrates a design space that is spanned by parameters of the compiler optimization, parameters of the architecture, and machine independent metrics.

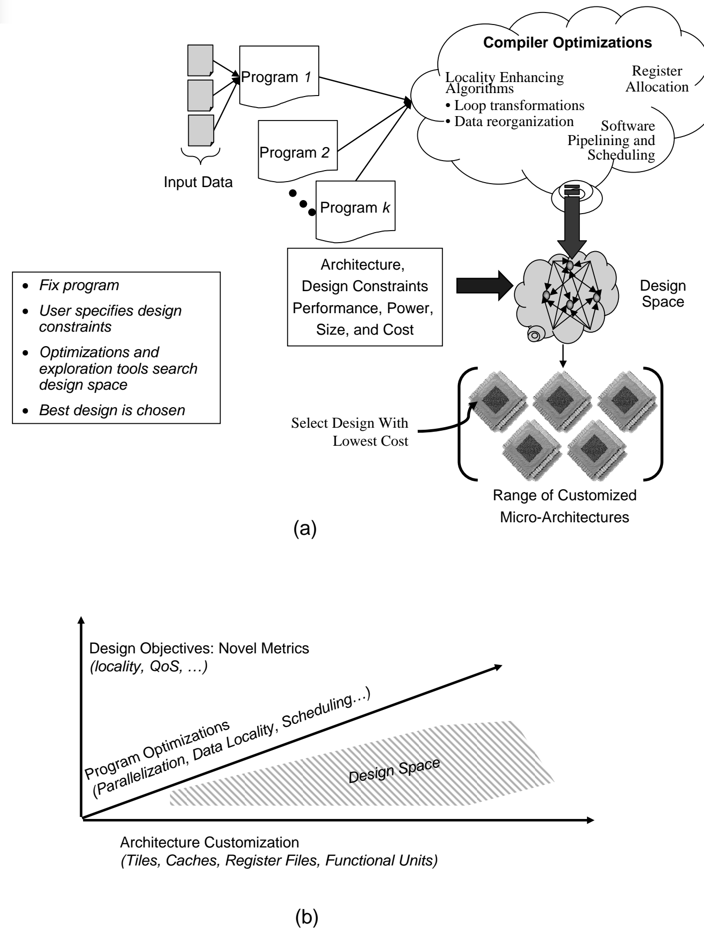

Fig. 2. A framework for compiler directed design space exploration

3.2 Specification of the Design Space

At its core, a design space explorer (DSE) accepts as input a program and a specification of the design space that includes optimization parameters, architecture parameters, a set of rules, and constraints. These rules and constraints may be derived from application execution constraints or domain-specific deployment cost and performance constraints. In the proposed approach the design-space is spanned by parametric representations of both candidate compiler optimizations and architecture parameters and the navigation of the design space is driven by quantifiable, machine independent metrics. While formal specification techniques for compiler optimizations has not drawn much attention, various specification mechanisms exist for describing architectures both for compilers, typically referred to as machine descriptions, as well for synthesis tools typically using hardware description languages (HDLs) such as VHDL or Verilog. While these

languages support the description of a single target architecture, additional constructs are necessary to describe a parameterized design space and associated constraints. A first step in this direction is PD-XML: a language for describing embedded processors and platforms [30]. The core PD-XML concepts are being extended by research efforts in CREST to specify the design spaces for configurable architectures along with their constraints.

3.3 Characterization and Exploration of the Design Space

The exploration of the design space utilizes characterizations of the behavior of the application as well as models of the target hardware. In the traditional compiler domain, program analysis and execution profiling have been the dominant approaches for characterizing application behavior in the context of fixed architectures. Though thorough hardware cost-performance trade-off studies have been difficult–this is an artifact of the target being fixed–application characterization using compiler analysis and profile-feedback have been instrumental for detecting optimization opportunities.

The implementation of candidate optimizations are guided by hardware models that quantify the impact of the optimizations, for example area and clock speed. Development of such hardware models is accompanied by decision models that guide the application of candidate program optimizations.

Such a framework that combines application characterization with high level hardware models serves as a point of departure for current generation design space exploration tools. Exploration itself can be carried out employing techniques ranging from heuristics to linear programming. Section 5 provides and example drawn from customization of the memory hierarchy and illustrates the use of machine independent metrics to characterize the demands of the application in the context of the memory hierarchy.

4 Attributes of the Design Space

Our framework for design space exploration has three main aspects: the space of target architectures, the space of compiler optimizations and the metrics that characterize each point in such a design space.

4.1 The Space of Target Architectures

The spectrum of target architectures for embedded systems range from fine grained field programmable gate arrays (FPGAs) [32] at one end, through coarse grained "sea of tiles" architectures (for example [34] [26] [12]) to traditional fixed-ISA microprocessors at the other end.

Fine grained FPGA architectures can implement complex logic and exploit fine grained logic-level parallelism but have very poor logic densities, delay, area and power consumption [36] relative to ASICs. Coarse grained "sea of tiles" architectures, though not as flexible as fine grained FPGAs greatly reduce power, area, delay and configuration times. In addition they map well to traditional compiler representations and abstractions of programs and hence serve as efficient compilation and design space exploration

targets. Further, compiler optimizations such as modulo scheduling can readily leverage the tiled nature of such architectures for efficient mapping of applications as well as to influence traditional EDA tasks such as placement and routing. More recently ”structured” ASIC design flows have emerged as a mechanism to reduce the cost of chip development [36]. Cost reductions are achieved by reducing the degrees of freedom in physical layout and optimization and by using pre-designed cores [3]. Tiled based architectures with their coarse grain structure and limited space of interconnect and placement options (relative to a full custom gate design) are a good fit for structured ASIC design flows. Most recently, a new class of coarse grained configurable architectures referred to as *polymorphic computing architectures* (PCAs) have emerged as a powerful class of architectures for satisfying demanding embedded computing requirements. PCAs represent a new class of computing architectures whose micro-architecture can *morph* to efficiently and effectively meet the requirements of a range of diverse applications thereby also amortizing development cost. Such architectures have the advantages of smaller footprint (by replacing several ASIC cores), lower risk via post deployment hardware configuration, and sustained performance over computationally diverse applications. This class of architectures is particularly amenable to compiler directed DSE. Finally, A-EPIC [22] and more recent efforts in customizable instruction set architectures [6] [5] [8] recognize the need for more flexible microprocessor targets and are amenable to compiler-directed DSE for custom instruction selection or configuration. In particular, powerful program analysis techniques can be brought to bear to ”discover” custom instructions. The challenge in discovering suitable custom instructions lay in the complex interactions instructions have with the data-path micro-architecture and the memory hierarchy.

Along this spectrum of target architectures, the application of compiler directed DSE is based on the ability to adequately characterize the hardware target via high level models utilized to navigate the design space. An additional crucial decision is the selection of the set of architectural parameters which form a design space that is self-contained (and hence can be characterized well) and thereby makes the process of design space exploration tractable. For example, target architectures range from conventional microprocessors to fine grained re-configurable targets like FPGAs. A subset of such target architectures which are easy to explore, self-contained as well as meet the customization requirements, will immediately leverage the benefits of compiler directed DSE.

4.2 The Space of Compiler Optimizations

Compiler optimizations can be broadly divided into two classes. Optimizations that influence the design of the memory subsystem and optimizations that influence the design of the data path of the embedded system. In the subsequent sections, we broadly sketch examples of optimizations in each class relevant to embedded systems customization.

Data Locality Optimizations In general, data locality optimizations can be classified into three classes. Code transformations which change the data object access pattern for a given data layout [9], data transformations that change the data object layout for a

given object access pattern [10] and optimizations which manage the memory hierarchy through techniques such as software based prefetching [33]. Among these techniques the last two are especially attractive in the context of memory hierarchy customization, as code transformation needs to be leveraged for co-optimizing the application and the data path of the target architecture. In this regard novel analysis techniques based on metrics like neighborhood affinity probability (NAP) [23] [20], characterize object access patterns in a program region specific manner, enable integrated optimizations that can drive data re-layout in memory as well as drive software based prefetching to reduce the memory hierarchy cost (in terms of area as well as power) while maintaining performance. Novel metrics also serve as an architecture independent measure of required memory hierarchy resources. Optimizations based on NAP not only reduce the cost of the memory hierarchy but also enable numerous trade-offs among various cache parameters.

Optimizations that improve data locality have been shown to improve performance in the context of conventional microprocessors. In addition, recent work [21] has shown how such data locality optimizations not only yield performance improvements in the case of fixed architectures, but can also be used for the design space exploration of memory hierarchies resulting in reduced costs, in terms of area and power, while maintaining the same performance as that of an unoptimized application. Metrics derived from profile-feedback and application analysis techniques introduced in previous studies [21] not only indicate optimization opportunities, but also serve as an indicator of required hardware resources.

Concurrency Enhancing Optimizations Traditional optimizations have focused on concurrency enhancement in the presence of fixed hardware resources. In the context of application-architecture co-optimization, since architectural resources are no longer fixed, concurrency in the application can be exploited to gain performance by investing in additional architectural resources. This is a natural way to trade off cost for performance. Several compiler optimizations ranging from fine grained ILP (Instruction level parallelism) techniques [16] to coarse grained loop parallelization techniques [35] can be applied to explore the cost (area) vs. performance of a variable number of functional units.

Scheduling Techniques Apart from data locality enhancing optimizations which influence the design of the memory hierarchy, and concurrency enhancing optimizations which influence the number of functional units, scheduling techniques can be leveraged to influence EDA steps such as placement and routing. For example, compile-time program analysis can be applied to characterize communications between functional resources and hence influence the placement (e.g., co-location) of these resources [28] for increased performance. In addition, program transformations such as software pipelining can be leveraged to perform integrated placement and routing of regular, systolic array-like structures [15].

Collectively, the classes of compiler optimizations described in the preceding can be leveraged to co-optimize the application with the memory hierarchy, and/or the datapath.

5 An Example: Memory Hierarchy Customization

In this section, we illustrate how a compiler can be leveraged for design space exploration to customize the memory hierarchy for a particular application. We also show how novel metrics, characterized through compiler analysis, aid in such an exploration and serve to quantify hardware resource needs. Formally the problem can be stated as follows: Given a cache with block size $\mathcal{B}$, size $\mathcal{S}$ and bandwidth $\mathcal{W}$ and a cost function $F : (\mathcal{B} \times \mathcal{S} \times \mathcal{W}) \rightarrow \Re$ and a program $\mathcal{P}$, what are the values of $\mathcal{B}$, $\mathcal{S}$ and $\mathcal{W}$ for the best performance such that the cost is less than a specified value c ?

To explore a design space, the program behavior should be characterized using architecture independent metrics, and the cost of a particular cache design and its impact on the program performance should be evaluated. The architecture independent metrics introduced in [20] completely characterize the intrinsic or virtual behavior of a program and relate the virtual characteristics to the realized or observed behavior when a micro-architectural model is imposed during execution. These metrics – *Turnover factor, Packing factor* and *Demand bandwidth* –are defined below:

Turnover Factor Intuitively, turnover factor can be defined as the amount of change in the working set of the program $\mathcal{P}$. Consider a data reference trace $\mathcal{T}$ for the program $\mathcal{P}$ as a string of addresses. $\mathcal{T}$ is partitioned into smaller non-overlapping substrings, $s_1, s_2, \cdots, s_n$. The number of unique characters in each substring s_i equals $\mathcal{V}$, the virtual working-set size. In addition, the substrings do not overlap, and therefore $\mathcal{T} = s_1|s_2|\cdots|s_n$. We now define the cost of a transition between two substrings s_i and s_{i-1} as the turnover factor.

$$\Gamma(s_{i>0}) = \mathcal{V} - |\hat{s}_i \cap \hat{s}_{i-1}|$$

where $\hat{s}_i$ is the set formed from s_i and $\hat{s}_0$ is the null set.

Packing Factor Packing factor can be loosely defined as the ratio of the total data in the cache to the useful data in the cache. Formally, given a mapping function $\mathcal{M}$ (derived from $\mathcal{B}$ and $\mathcal{S}$) which maps addresses in the trace $\mathcal{T}$ to blocks, let $\mathcal{R}_i$ be the number of blocks needed to map every address $k \in \hat{s}_i$. Now the packing factor can be defined as

$$\Phi(s_i) = \frac{\mathcal{R}_i \times \mathcal{B}}{\mathcal{V}}$$

This measures the efficiency with which useful data is packed into the cache.

Demand Bandwidth The demand bandwidth of an application can be defined as

$$\mathcal{D}(s_{i>0}) = \frac{\Gamma(s_i) \times \Phi(s_i)}{|s_i|}$$

Abstractly, it captures the rate at which data should be delivered to the caches to meet the program requirements. If the architectural bandwidth does not match the demand bandwidth, the program incurs stalls.

In light of these novel metrics, it is clear that a low packing factor and hence low demand bandwidth is desirable and has a direct impact on program performance. Low packing factor indicates optimal use of the caches and hence better cache design. For a given trace $\mathcal{T}$, packing factor is in turn determined completely by the block size $\mathcal{B}$, and cache size $\mathcal{S}$. Given an application, an automatic design space explorer can generate traces and then optimize the block size, cache size and bandwidth of the cache by calculating the packing factor and hence demand bandwidth for various configurations. Cost for each of these configurations can also be calculated through the cost function F and can be traded off for performance, without the need for time consuming cycle accurate simulation at each point of the design space. .

6 Remarks and Conclusions

A central impediment to the continued growth of embedded systems are the twin hurdles of NRE costs and time-to-market pressures. To address these challenges we need fundamental shifts in principles governing hardware-software design that can lead to the major leaps in designer productivity and thence cost reductions to sustain the performance and cost benefits of Moore's Law for future custom embedded systems. In this paper we describe compiler directed design space exploration (DSE) where the design space is spanned by parametric representations of both candidate compiler optimizations and architecture parameters and the navigation of the design space is driven by quantifiable, machine independent metrics. DSE in conjunction with emerging configurable architectures such as coarse grain tiled architectures, microprocessors with customizable ISAs, polymorphic computer architectures, and fine grained FPGAs, offer opportunities for concurrently optimizing the program and the hardware target thereby leveraging optimizing compiler technology with EDA design techniques. We expect such cross-fertilization between EDA and compiler technology will become increasingly important and become a source of significant improvements in designer productivity and system performance.

References

1. Celoxica: http://www.celoxica.com/.
2. Impulse accelerated technologies: http://www.impulsec.com/.
3. LSI logic rapid chip platform ASIC: http://www.lsilogic.com/products/rapidchip_platform_asic/.
4. Quicksilver technology: http://www.qstech.com/.
5. Stretch inc. http://www.stretchinc.com/.
6. Tensilica: http://www.tensilica.com/.
7. M. Baleani, F. Gennari, Y. Jiang, R. B. Y. Patel, and A. Sangiovanni-Vincentelli. HW/SW partitioning and code generation of embedded control applications on a reconfigurable architecture platform. *Proceedings of the tenth international symposium on Hardware/software codesign*, pages 151 – 156, 2002.
8. M. Baron. Stretching performance. *Microprocessor Report*, 18, Apr. 2004.
9. S. Carr, K. S. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, 1994.

10. T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. *Proceedings of the ACM SIGPLAN 99 Conference on Programming Language Design and Implementation*, 1999.
11. R. Goering. C design goes soft: EETimes: http://www.eetimes.com/story/oeg20010423s003, Apr. 2001.
12. C. R. Hardnett, A. Jayaraj, T. Kumar, K. V. Palem, and S. Yalamanchili. Compiling stream kernels for polymorphous computing architectures. *The Twelfth International Conference on Parallel Architectures and Compilation Techniques*, 2003.
13. H. Jones. How to slow the design cost spiral. *Electronic design Chaing: http://www.designchain.com*, Sept. 2002.
14. J. Liao, W. Wong, and M. Tulika. A model for hardware realization of kernel loops. *Proc. of 13th International Conference on Field Programmable Logic and Application, Lecture Notes of Computer Science*, 2778, Sept. 2003.
15. B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. *Design, Automation and Test in Europe Conference and Exhibition*, 2003.
16. W. mei W. Hwu, R. E. Hank, D. M. Gallagher, S. A. Mahlke, D. M. Lavery, G. E. Haab, J. C. Gyllenhaal, and D. I. August. Compiler technology for future microprocessors. *Proceedings of the IEEE*, 83(12), Dec. 1995.
17. K. Palem, S. Talla, and W. Wong. Compiler optimizations for adaptive epic processors. *First International Workshop on Embedded Software, Lecture Notes of Computer Science*, 2211, Oct. 2001.
18. K. V. Palem. C-based architecture assembly supports custom design: EETimes: http://www.eet.com/in_focus/embedded_systems/oeg20020208s0058, Feb. 2002.
19. K. V. Palem, D. S. Fussell, and A. J. Welch. High level optimization in a silicon compiler. *Department of Computer Sciences, Technical Report No 215, University of Texas, Austin, Texas*, Nov. 1982.
20. K. V. Palem and R. M. Rabbah. Data remapping for design space optimization of embedded cache systems. *Georgia Institute of Technology, College of Computing Technical Report*, (GIT-CC-02-10), 2002.
21. K. V. Palem, R. M. Rabbah, V. M. III, P. Korkmaz, and K. Puttaswamy. Design space optimization of embedded memory systems via data remapping. *Proceedings of the Languages, Compilers, and Tools for Embedded Systems and Software and Compilers for Embedded Systems (LCTES-SCOPES)*, June 2002.
22. K. V. Palem, S. Talla, and P. W. Devaney. Adaptive explicitly parallel instruction computing. *Proceedings of the 4th Australasian Computer Architecture Conference*, Jan. 1999.
23. R. M. Rabbah and K. V. Palem. Data remapping for design space optimization of embedded memory systems. *In ACM Transactions on Embedded Computing Systems (TECS)*, 2(2), 2003.
24. B. R. Rau and M. Schlansker. Embedded computing: New directions in architecture and automation. *HP Labs technical report: HPL-2000-115*, 2000.
25. A. Sangiovanni-Vincentelli and G. Martin. A vision for embedded software. *Proceedings of the International Conference on Compilers, Architectures and Synthesis of Embedded Systems*, pages 1 – 7, 2001.
26. K. Sankaralingam, R. Nagarajan, H. Liu, J. Huh, C. Kim, D. Burger, S. Keckler, and C. Moore. Exploiting ilp, tlp, and dlp using polymorphism in the trips architecture. *30th Annual International Symposium on Computer Architecture (ISCA)*, 2003.
27. S.-B. Scholz. Single assignment c – efficient support for high-level array operations in a functional setting. *Journal of Functional Programming*, 13(6), 2003.

28. R. Schreiber, S. Aditya, S. Mahlke, V. Kathail, B. R. Rau, D. Cronquist, and M. Sivaraman. Pico-npa: High-level synthesis of nonprogrammable hardware accelerators. *HP Labs technical report: HPL-2001-249*, 2001.
29. R. Schreiber, S. Aditya, B. R. Rau, V. Kathail, S. Mahlke, S. Abraham, and G. Snider. High-level synthesis of nonprogrammable hardware accelerators. *HP Labs technical report: HPL-2000-31*, 2000.
30. S. P. Seng, K. V. Palem, R. M. Rabbah, W.-F. Wong, W. Luk, and P. Cheung. Pd-xml: Extensible markup language for processor description. *In Proceedings of the IEEE International Conference on Field-Programmable Technology (ICFPT)*, Dec. 2002.
31. B. So, M. W. Hall, and P. C. Diniz. A compiler approach to fast hardware design space exploration in fpga based systems. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2002.
32. S. Talla. *Adaptive Explicitly Parallel Instruction Computing*. PhD thesis, New York University, Department of Computer Science, 2000.
33. S. VanderWiel and D. J. Lilja. Data prefetch mechanisms. *ACM Computing Surveys (CSUR)*, 32(2), June 2000.
34. E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: Raw machines. *IEEE Computer*, Sept. 1997.
35. M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *Proceedings of the IEEE Transactions on Parallel and Distributed Systems*, 2(4), Oct. 1991.
36. B. Zahiri. Structured ASICs: Opportunities and challenges. *21st International Conference on Computer Design*, 2003.