Model-Integrated Design Toolset for Polymorphous Computer-Based Systems

Brandon Eames
Vanderbilt University
b.eames@vanderbilt.edu

Ted Bapty
Vanderbilt University
bapty@isis.vanderbilt.edu

Ben Abbott
Southwest Research Institute
babbott@swri.edu

Sandeep Neema
Vanderbilt University
neemask@isis.vanderbilt.edu

Kumar Chhokra
Vanderbilt University
kg.chhokra@vanderbilt.edu

Abstract

Polymorphous computer-based systems are systems in which the CPU architecture "morphs" or changes shape to meet the requirements of the application. Optimized and efficient design for these systems requires exploration along axes beyond those of traditional system design. In this paper we outline a model-integrated toolset to aid in the specification, analysis and synthesis of polymorphous applications.

Polymorphous systems can be developed utilizing a four-tiered approach, where inherent application properties and characteristics govern design practices at each level. We show through the development of the model-integrated approach that polymorphous system design is inherently coupled with the search and exploration of a combinatorial space of design tradeoffs. Design tools are needed to efficiently evaluate this large and complex space in order to arrive at near-optimal application implementations.

1. Introduction

Embedded systems are computer based systems that are deployed into the environment, in weapons, factories, communication devices, medical devices, etc., far from the usual computer room. They are typically very resource limited. These systems interface to sensors, extract information from large volumes of raw data, and make complex decisions based on the information gathered from the environment. Achieving a balance between the minimization of resources and implementing the necessary capabilities requires efficient implementations.

The time and resources required to execute an algorithm or component on a particular platform is not purely based on clock speed of the underlying hardware. Rather, the inherent mapping of the component to the hardware shape often plays a dramatic role in utilization. Certain types of computations perform better on particular platforms. Thus, various classes of computers have evolved. Take for example, SuperScalar (Intel), DSP (TI), Java Chips (Sun), VLIW (Multiflow), Lisp Processors (Symbolics), Vector (Cray), etc. Performance in this sense can have many different metrics, be it actual execution time, throughput, power consumption, latency, etc. Given a particular metric, a component's performance can be evaluated across several different platforms, and the best platform selection can be determined. If all components in a system were allowed to execute on their optimal platform, ignoring compositional consequences, the system as a whole would exhibit the best performance according to that metric. This affinity between components and platforms is a property of a component, and forms the basis for a polymorphous embedded system design.

Recent advances in VLSI technology have allowed chip developers to create a new class of computer architectures. These new architectures are designed to natively support multiple modes of computation. Examples of such architectures include Smart Memories [10], Monarch [6], RAW [19], and Cyclops [4]. Hardware support for multiple computation modes makes this emerging class of architectures an ideal candidate for exploiting the affinity between computations and compute platforms. However, there exists a large void in the area of tool support for developing applications for these architectures. Architecture teams are developing low-level tool support, such as compilers and runtime systems. However, the reasoning required to analyze and optimize multimodal polymorphous systems must be much deeper than what current compilation technology is capable of. Advanced tool support with capabilities to reason across the full application and architecture space is required in order to develop near-optimal software targeting polymorphous hardware.

In the embedded system domain, achieving performance is critical, along with the performance guarantees required for real-time systems. A polymorphous approach to system design can achieve large increases in performance, at the cost of complexity in design and implementation. The use of sophisticated design tools can mitigate not only the complexity involved in computer-based system design, but the increased complexity involved in polymorphous system design as well.

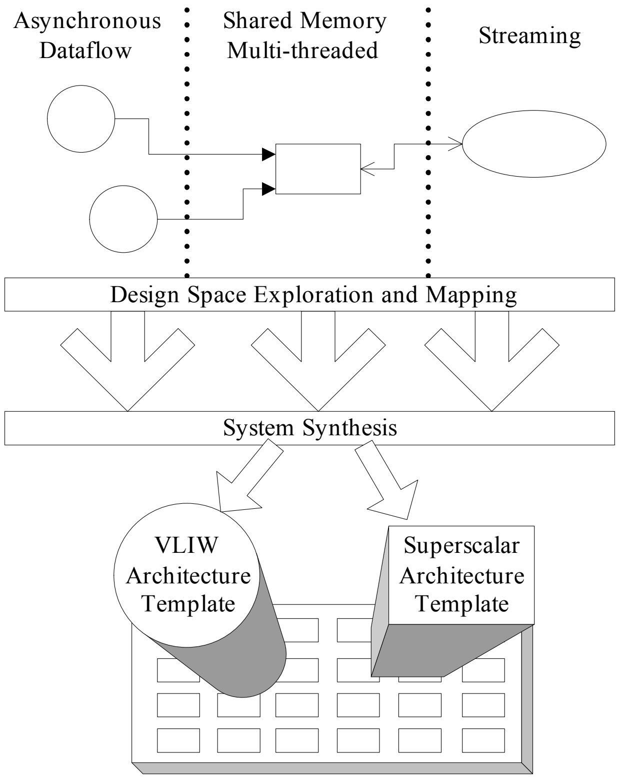
Asynchronous Dataflow
Shared Memory Multi-threaded
Streaming
Design Space Exploration and Mapping
System Synthesis
VLIW Architecture Template
Superscalar Architecture Template

Figure 1 . Polymorphous approach to embedded system design

This paper presents an overview of a model-integrated toolset which facilitates complex polymorphous application development. Figure 1 illustrates the role of the toolset. As complex applications are composed of heterogeneous components, the tool supports component capture via models of computation, such as Synchronous Dataflow [13] or shared memory multithreading. Design space exploration evaluates design tradeoffs, reasoning across the full application and architecture space, and searches for small sets of near-optimal design implementations. Resource allocation is addressed through design-space exploration. System synthesis generates functional implementations directly from system models. The target platform for a polymorphous design is a configurable polymorphous architecture, which facilitates via hardware support the execution of applications targeting multiple specialized architectures. To allow the configuration of polymorphous hardware, the tools support a concept called an architecture template. An architecture template is a predetermined configuration of the underlying polymorphous hardware. Software is developed for a particular architecture template, and then automatically mapped down to the polymorphous architecture.

The remainder of the paper details the various aspects of the toolset. In section 2, we discuss the model-integrated approach to system development and how it applies to polymorphous system design. Section 3 addresses the concept of architecture templates. Section 4 discusses the design space exploration problem, followed by section 5, which addresses system synthesis. Section 6 shows a brief example of the tools applied in the polymorphous design of a speech recognition engine. Section 7 discusses related work and section 8 summarizes the paper.

2. Model Integrated Computing and Polymorphous System Design

Model Integrated Computing (MIC) [18] is an approach to designing complex computer-based systems. Models capture system design information, environmental interactions, and other constraints on system composition. The models are composed in a customized, multi-aspect, domain-specific language. Specialized software generators, called Interpreters, traverse these models to extract design information, make decisions on system implementation, and synthesize code, analyses, simulations, and other tools used in building and verifying the system. MIC has been successfully applied in several domains [2][5][8][12].

Here, we apply MIC techniques to mitigate the complexity of developing embedded applications for polymorphic architectures. Such applications are composed of many heterogeneous components or subsystems. Heterogeneity arises from the differing nature of the algorithms employed in the subsystems, as well as how the algorithms compose. The most natural way of capturing and characterizing components differs not only from application to application, but from component to component within complex applications. Models of computation (MoC) represent a set of formal modeling semantics for capturing various classes of components. We use multiple models of computation as a semantic basis for domain-specific modeling languages in the MIC environment. The modeling environment provides a framework for establishing interactions between the models of computation. This approach is similar to the approach taken in the Ptolemy project [14]. The basic models of computation we support are:

- Synchronous streaming model [11], allows efficient specification of multimedia operations, where predictable data patterns and interlocking computations can be fully specified.
- Asynchronous Dataflow, where system computations must be driven by availability of data, and computations are ordered in a specified computational graph. See Figure 2.
- Multithreaded, Shared Memory, where computations operate concurrently, communicating via shared memory and synchronization primitives, as in POSIX threads (Pthreads). This MoC can be extended for distributed memory operations, with communications operations, such as MPI. See Figure 3.

Models of computation facilitate efficient design capture by using formalisms that match the problem. For example, synchronous, data parallel programs are very efficiently expressed using a streaming model of computation. The computer-based system application domain integrates heterogeneous classes of computations; therefore we support multiple models of computation to efficiently and cleanly model each class. Efficient component capture allows the generation of efficient implementations directly from the captured design. By supporting multiple models of computation, the modeling environment allows the capture of the kinds of components which span the space of components required by complex embedded system.

Figure 2 shows an example of a system modeled using the dataflow MoC. Boxes represent functions, or blocks of code which operate on data. The boxes have arbitrary numbers of inputs and outputs, which are the input tokens (parameters) and output tokens (results) of the function. Connections between boxes represent data dependencies or paths through which data are sent from one component to another. Data connections are implemented as queues. Data flows through the system in the form of tokens. When a process or actor executes, it consumes tokens from its input queues and enqueues tokens in its output queues. Dataflow is a common model of computation used in the signal processing community, and a great deal of research has occurred in specifying, analyzing, and implementing the dataflow MoC.

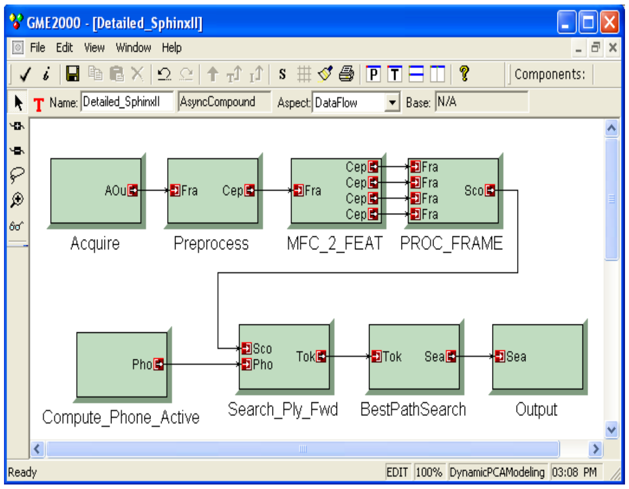

Figure 2 . Model of a speech recognition algorithm expressed in dataflow semantics

Figure 3 illustrates the modeling syntax of the shared-memory multithreaded modeling language. Boxes represent threads, with lines representing dependencies between threads, implemented as thread synchronizations.

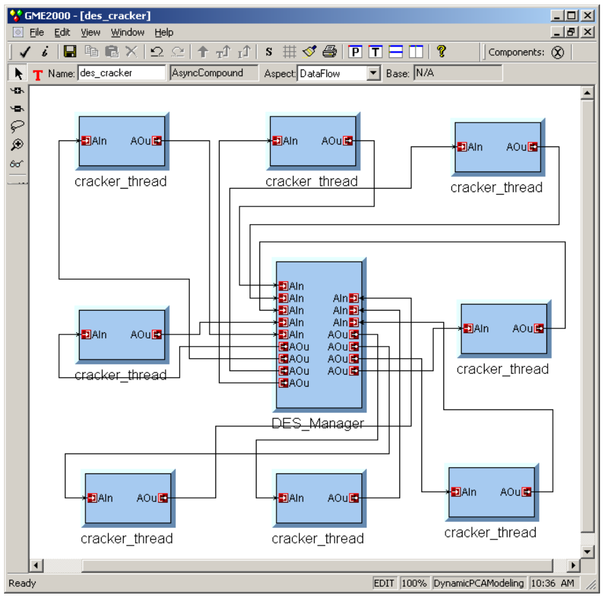

Figure 3 . Model showing multithreaded modeling language syntax

As heterogeneous applications are composed of components which are best modeled using distinct models of computation, the toolset supports the interaction of component implementations at runtime. The runtime system provides memory management support through buffers. How individual components utilize data depends on the model of computation used in developing the component. For example, in a synchronous streaming model of computation, data tokens or records are packed together in a buffer, and are accessed as a stream. Individual records from a stream can be distributed to stream kernels which process data in a data parallel fashion. The results are collected and packed into a stream buffer. Such a stream buffer can be processed by a component implemented in a dataflow model of computation simply by viewing the stream buffer as a collection of dataflow tokens. Records are extracted from the stream buffer one record at a time and enqueued into the appropriate input queues of a component one token at a time. The dataflow component simply extracts these tokens when they are available. Dataflow components can send data to a streaming component simply by having the output tokens of a dataflow component collected and packed into a stream buffer.

In a similar vein, streaming components can interact with components implemented using the multithreaded model of computation. A stream buffer can be viewed as a shared memory location whereby multiple threads of execution can extract records as needed. Similarly, a shared memory buffer written to by a multithreaded component can be re-packed into a stream buffer for access by a streaming component. Dataflow components can interact with multithreaded components following similar semantics.

3. Architecture Templates

Many years of effort in the fields of computer architecture research and associated compiler support have been invested in search of optimizing computer performance assuming specific classes of problems [7]. The last three decades have seen significant development in techniques to exploit fine-grained or instruction-level parallelism (ILP). Examples of successful architectures are VLIW, superscalar, vector, SIMD, etc. Previously, we discussed the affinity between algorithms and architectures, characterizing the occurrence of significant performance gains when algorithms map well to architectures. With the extreme heterogeneity exhibited in complex embedded applications, it is not possible to select a single architecture class as an implementation platform for an application, and achieve high affinity between the platform and the application's constituent algorithms.

Polymorphous computing can emulate multiple architectures in the same device. Given this capability, designers can tailor component implementations to those architectures to which the algorithms exhibit the strongest affinity. These tailored implementations must then be translated and mapped to an underlying polymorphous architecture by the synthesis tools. These tools require formal structuring information, in architecture models. We refer to these architecture models as ***architecture templates***.

An architecture template serves two purposes. First, it provides a framework for leveraging proven techniques for exploiting fine-grained parallelism. Second, it provides a means of mapping applications to a complex, configurable platform. Preliminary techniques for selecting architecture templates for a polymorphous architecture are under development (described in the Design Space Exploration section below). The architecture template models allow the user to assert hints and constraints to assist the design space mapping. Architecture templates allow existing compiler technology to be leveraged in translating algorithms to machine code for the polymorphous platform. Template-specific compilation translates code to an intermediate form, which is then mapped down to the polymorphous architecture using hints provided as part of the template definition.

An architecture template captures a class of physical computer architectures. We use parameterized modeling to represent variability within an architecture class. For example, a VLIW has parameters such as number of functional units, e.g. 4-wide VLIW or 8-wide VLIW. Parameters in architecture templates allow the template to be tuned to best exploit the performance characteristics of an algorithm. Other example parameters for various templates are: length of vector for vector template; issue width for VLIW template, issue width for superscalar template.

The hierarchical system composition using Models of Computation permit a fine-grained component representation. These fine-grained components represent the fundamental computational algorithms and building blocks of the system. We associate fine-grained components with architecture templates, and then map the association onto polymorphous hardware. A visual representation of an architecture template is shown in the context of an example application in Section 6 (see Figure 5).

4. Exploration of a Polymorphous Design Space

Classic system design involves searching and trading between various tradeoffs. For example, if a particular architecture is chosen, then what clock speed and power will be necessary to avoid missing system deadlines? Polymorphous systems add a further dimensions of complexity to this search space, multiple and dynamic architecture shapes. Accordingly, successful polymorphous system design involves many tradeoff decisions. There are decisions to be made at the level of application composition, such as what type of algorithm should be used to perform a particular task (i.e. spatial vs. spectral filtering of an image). As applications are resolved into collections of fine-grained components, decisions must be made about associations with architecture templates. Further, as each architecture template is parameterized, an optimal set of values for each parameter must be obtained for each component-template association. Finally, how architecture template-component pairings must be allocated actual hardware resources on the underlying polymorphous architecture. All of these tradeoff decisions must be analyzed in the context of global application requirements or goals. If an application is required to consume less than 10 watts of power, all tradeoff decisions (or perhaps only those which are relevant) must be evaluated under this light.

We call the space of alternative implementations formed by every possible combination of tradeoff decisions a design space. Formally we can define the design space in the following manner.

Let, A_i be the set of alternatives for an application component i, and let N_c be the number of components in the application. Then, we can define the application space AS as the cross-product set:

$$AS = \prod_{i}^{N_c} A_i$$

Now, let D_j be the domain of parameter j, and let P_k be the set of parameters in a parameterized architecture template k. Then, we can define the set of possible instantiations PS_k of the architecture template k as:

$$PS_k = \prod_{j}^{P_k} D_j$$

In this formalism, we consider a polymorphous computer as a collection of two distinct types of unit resources, a) shareable –multiple architecture templates instantiated on the polymorphous computer can simultaneously be assigned these units e.g. floating-point unit, caches, etc. on the IBM-Cyclops reference architecture [4], and 2) non-shareable – each architecture template instantiated on a polymorphous computer is uniquely assigned these units, e.g. thread unit on Cyclops. Thus, here we define a polymorphous computer PC simply as:

$$PC = R_s \cup R_{ns}$$

where, R_s is the set of shareable resource units, and R_{ns} is the set of non-shareable resource units.

Based on this definition of a polymorphous computer, we can define the possible instantiations of an architecture template on a polymorphous computer, in terms of the resource requirements of an architecture template instance t_m and its mapping on the resource units in the polymorphous computer PC. Let, $S(t_m)$ and $NS(t_m)$ be the shareable and non-shareable resource requirements, respectively, of $t_m \in PS_k$. Then, the set of possible mappings $MS(t_m)$ of t_m is defined as:

$$MS(t_m) = \begin{pmatrix} \|R_s\| \\ \|S(t_m)\| \end{pmatrix} \times \begin{pmatrix} \|R_{ns}\| \\ \|NS(t_m)\| \end{pmatrix}$$

The set of all mappings MS_k of architecture template k, can now be defined as:

$$MS_k = \bigcup_{\forall t_m \in TS_k} MS(t_m)$$

For implementing an application the polymorphous computer can be partitioned to accommodate zero or more instances of each architecture template k. If we let L_k be the number of instances of architecture template k, then we can define the possible set of partitions and mappings of the polymorphous computer as:

$$RS = \prod_{k} \prod_{j}^{L_k} MS_j$$

We designate the set RS as the resource space, because it captures all possible ways in which the polymorphous computer can be configured to deploy an application. Note that L_k is bounded such that the sum of non-shareable resource units required by all architecture template instances is less than $\|R_{ns}\|$.

The design space for the system is then defined as:

$$DS = AS \times RS$$

We call the process of evaluating tradeoffs across this space design space exploration. Based on the above definition it is easy to observe that the size of the design space can be extremely large. Some preliminary work in developing the speech recognition system described in Section 6 indicate a design space on the order of 10^25 alternative design implementations. Clearly, an enumerative search through this space for feasible and near optimal solutions will be prohibitively expensive. Therefore, we require efficient and scalable search techniques to rapidly evaluate the design space for feasible solutions.

In prior research, Neema [15] has developed a tool to address design space exploration. In his approach, non-functional requirements of the system are viewed as constraints, and the process of exploring the design space amounts to a constraint satisfaction problem. He used Ordered Binary Decision Diagrams [3] to symbolically represent the design space, and encoded constraints as operations on OBDD's to efficiently explore the space.

The application of a constraint amounts to pruning out designs invalid with respect to the applied constraints, and produces a much smaller space, from which designs can be chosen for synthesis, or the reduced space may be further explored with finer-grained performance analysis tools and simulation. Neema did not attempt to search parametric design spaces with his approach. Parametric spaces do not map cleanly onto the OBDD approach, since it requires a large number of binary variables to encode the domain of parameters, and may result in "exponential blow-up" in the computational complexity.

In this research we are investigating other techniques for efficient design space exploration, and their efficient integration into the OBDD-based symbolic constraint satisfaction approach. Techniques under consideration include integer linear programming using branch and bound, genetic algorithms, constraint logic programming, and simulated annealing.

5. System Synthesis

System synthesis is the process of converting a point in the design space into a physically realizable implementation. This process entails generating all of the necessary artifacts for:

- Configuring architectures – the parameters to make the architecture morph into VLIW, MIMD, etc, or implementing a specific communication topology. This can include code sequences, tables, memory controller maps, network switch settings, etc.
- Configuring the middleware (morphware) for the application and architecture. This can include the link tables of required OS facilities, static schedules for processors, message routing tables, DMA engine sequences, etc.
- Generating the application software, composing the software from libraries and glue code. Architecture-efficient implementations are pulled from libraries. Interface code for communication and synchronization must be generated and inserted as wrappers around the components. Interfaces to access the morphware facilities must also be created and integrated.

6. Polymorphous Design of Speech Recognition Software

We are applying our approach to polymorphous system design in the development of a speaker-independent speech recognition system, based on Sphinx [17]. Figure 2 shows the top-level view of the Sphinx application, utilizing dataflow semantics. Sphinx takes sampled audio data, applies some signal processing algorithms on the front end, collects samples into 10ms speech frames, and then passes those speech frames on to a recognition engine. The recognition engine generates feature vectors for each frame, and then compares those feature vectors against several pre-computed vectors characterizing basic acoustic sounds in the English language. The best score for each feature is calculated and the result is passed to a Hidden Markov Model search engine, which attempts to piece together words from simple sounds. For simple vocabularies, system execution time is dominated by the calculations to compare feature vectors (a distance calculation), and to discover highest scores of distances. Figure 4 shows a hierarchical decomposition of the Proc_Frame component. The Proc_frame component is responsible for calculating distances between feature vectors, as well as scoring the results of the computations.

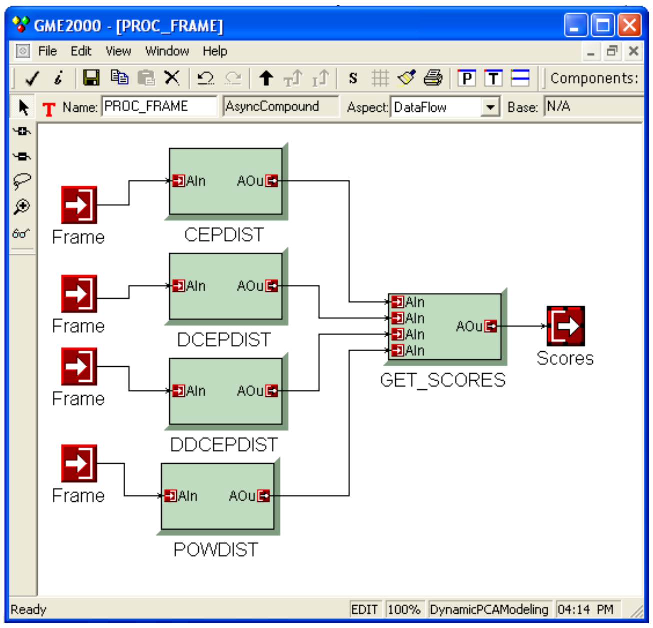

Figure 4 . Proc_Frame component of the Sphinx application

An examination of the Get_scores component reveals a very static structure, tending towards a VLIW implementation. The Get_scores component merges the results of the distance calculations across the four features. It performs similar, data independent operations for each of the four feature types, across a large array of distances. It forms a merged distance score for each speech frame. An analysis of the available fine-grained parallelism in the get_scores component leads us to associate the component with a VLIW template, with a four instruction issue width. Figure 5 depicts the representation of this association, as well as a definition of the four-issue VLIW template.

A template is captured as a mapping between the logical architectural concepts of the template and the physical architecture resources of the underlying polymorphous computer. As seen in Figure 4, in this example we capture the IBM Cyclops [4] reference architecture as a basic polymorphous computer. We show only one tile of the Cyclops architecture. The VLIW template view shows how logical concepts map onto the architectural features of the Cyclops architecture. For example, the logical single threaded instruction stream of the VLIW architecture maps onto four separate thread execution units of the Cyclops architecture. This resource mapping provides information to the resource allocation stage of design space exploration.

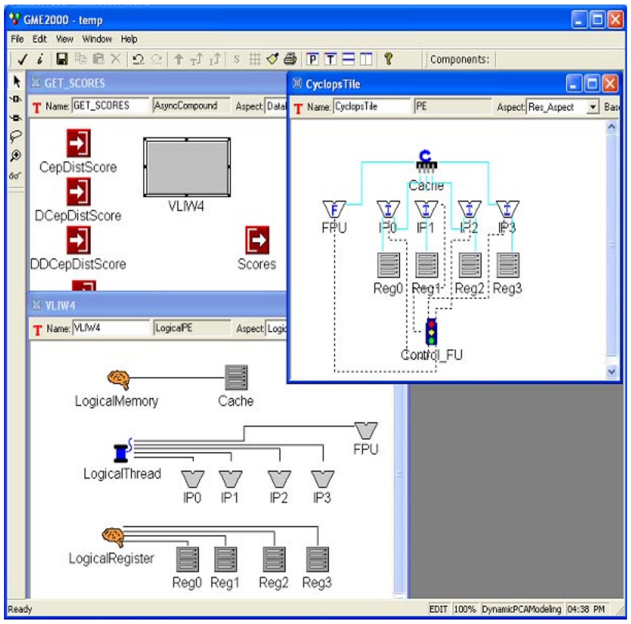

Figure 5 . Models representing the Getscores fine-grained component, with a VLIW4 template applied to a Cyclops hardware architecture

After the application is fully modeled, and associations are made between fine-grained components and architecture templates, the full system is synthesized from the models, and executed on the final platform. At this stage of the tool development, associations between fine-grained components and architecture templates must be made explicitly. As part of the design space exploration toolset, we are developing algorithms to automate this decision process. The output of the synthesis stage is glue code targeting runtime middleware (morphware) to support inter-component communication and synchronization.

7. Related Work

While we believe our approach to polymorphous system development to be unique, there are many related research areas, and we leverage work from several previous projects as well.

In prior work at Vanderbilt University, the ACS toolset [2] was developed to synthesize embedded applications targeting heterogeneous processing platforms including FPGAs, DSPs, and general purpose processors. Synthesis of component-based software was addressed.

The Ptolemy project [14] at UC Berkeley researches models of computation and semantic interactions of models of computation. A goal of the Ptolemy project is to allow the utilization of those models of computation which best fit the problem, and to facilitate the interaction of components modeled in distinct models of computation.

Design space exploration as a research topic has received much attention in the System-on-Chip design tool community, with the goal of finding optimal or near-optimal hardware-software partitions [1], or of configuring parameterized hardware components for a particular application [16]. One such project is the PICO project[9] under development at HP Labs. PICO seeks to synthesize custom VLIW-based processors which are tailored for a particular application or class of applications. Parameterized models are searched using a tool called the Spacewalker to determine optimal settings for a particular application class.

While these search techniques improve space exploration times dramatically over exhaustive search techniques, they are still too expensive to apply to combinatorial spaces such as the types observed in polymorphous system design. However, hybrid techniques involving fast OBDD-based searches and parametric design space search techniques can be developed to improve the overall search coverage and search time.

8. Conclusions

Polymorphic system design is a novel approach to system development, offering an opportunity to achieve new levels of performance and efficiency. This potential efficiency gain comes at the cost of increased system design complexity. Sophisticated design tools are required to support the development of polymorphous applications, due to the complex interactions between subsystems and the increased number of design variables which must be considered.

We have described the model-integrated toolset, under development at ISIS, which supports the development of high-performance, near-optimal polymorphous systems. The tool allows developers to express software at a high level of abstraction, independent of the underlying architecture. Generation tools navigate the design space, resolving design issues, such as architecture selection, choosing appropriate implementation options, mapping to resources, etc.. The tools automate the process of designing near-optimal system implementations, using information captured in the models, design-space navigation techniques, and software generation methods.

9. Acknowledgements

This work was supported under the DARPA PCA program, under contract F30602-01-C-0078. The authors wish to thank the members of the M3T team for their inputs to this work.

10. References

[1] Azzedine A., Diguet J., and Philippe J. "Large Exploration for HW/SW Partitioning of Multirate and Aperiodic Real-Time Systems", Proceedings of the 10th International Symposium on Hardware/Software Codesign, Colorado, May, 2002.

[2] Bapty T., Neema S., Scott J., Sztipanovits J., Asaad S. "Model-Integrated Tools for the Design of Dynamically Reconfigurable Systems", VLSI Design, 10, 3, pp. 281-306, 2000.

[3] Bryant R., "Symbolic Manipulation with Ordered Binary Decision Diagrams," School of Computer Science, Carnegie Mellon University, Technical Report CMU-CS-92-160, July 1992.

[4] Cascaval, C. el al. "Evaluation of a Multithreaded Architecture for Cellular Computing." 8th International Symposium on High-Performance Computer Architecture, Feb 02-06 2002. Boston MA.

[5] Davis J., Scott J., Sztipanovits J., Martinez M.: Multi-Domain Surety Modeling and Analysis for High Assurance Systems, Proceedings of the Engineering of Computer Based Systems, pp. 254-260, Nashville, TN , March, 1999.

[6] Granacki, J. and Vahey, M. "Monarch: A High Performance Embedded Processor Architecture with Two Native Computing Modes." *High Performance Embedded Computing Workshop*, September 2002.

[7] Hennessy J. and Patterson D. Computer Architecture: A Quantitative Approach, 2nd Edition. Morgan Kauffman Inc, San Francisco CA, 1996.

[8] Karsai G., DeCaria F. "Model-Integrated On-line Problem-Solving Environment for Chemical Engineering", IFAC Control Engineering Practice, 5, 5, pp. 1-9, 1997.

[9] Kathail V., Aditya S., Schreiber, R., Rau B. R., Cronquist D., Sivaraman M. "PICO: Automatically Designing Custom Computers" IEEE Computer, September 2002, pp. 39-47.

[10] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, M. Horowitz. "Smart Memories: A Modular Reconfigurable Architecture." *International Symposium on Computer Architecture*, June 2000.

[11] Peter Mattson, *Programming System for the Imagine Media Processor*, Ph.D. thesis, Stanford University, 2002.

[12] Misra A., Karsai G., Sztipanovits J., Ledeczi A., Moore M. "A Model-Integrated Information System for Increasing Throughput in Discrete Manufacturing", International Conference and Workshop on Engineering of Computer Based Systems, pp 203-210, Monterey, CA, March 24, 1997

[13] E. A. Lee and T. M. Parks, "Dataflow Process Networks,", *Proceedings of the IEEE*, vol. 83, no.5, pp. 773-801, May, 1995.

[14] Edward A. Lee, "Overview of the Ptolemy Project," *Technical Memorandum UCB/ERL M01/11*, University of California, Berkeley, March 6, 2001.

[15] Neema, S. "Design Space Representation and Management for Model-Based Embedded System Synthesis." Technical Report # ISIS-01-203. Vanderbilt University, February 2001.

[16] Palesi, M. and Givargis, T. "Multi-Objective Design Space Exploration Using Genetic Algorithms", Proceedings of the 10th International Symposium on Hardware/Software Codesign, Colorado, May, 2002.

[17] M. Raveshankar. "Efficient Algorithms for Speech Recognition." Ph.D. thesis, Carnegie Mellon University, 1996.

[18] Sztipanovits J., Karsai G. "Model-Integrated Computing", IEEE Computer, pp. 110-112, April, 1997.

[19] 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, September 1997, pp. 86-93.