Programming Heterogeneous Hardware Components with Software Components

Hadi Esmaeilzadeh
University of Washington
hadianeh@cs.washington.edu

Sylvain Girbal
Thales TRT, France
sylvain.girbal@thalesgroup.com

Kathryn S. McKinley
The University of Texas at Austin
mckinley@cs.utexas.edu

Olivier Temam
INRIA Saclay, France
olivier.temam@inria.fr

Sami Yehia
Thales TRT, France
sami.yehia@thalesgroup.com

Abstract

Due to performance and power scalability constraints, a likely path forward for architectures is distributed-memory heterogeneous chip multiprocessors (CMPs) composed of a mix of cores and accelerators. Regardless of whether the architectures are heterogeneous or homogeneous CMPs, efficiently programming such architectures is a daunting challenge. We propose a pragmatic programming approach for non-expert users to make it easier for them to extract performance from homogeneous and heterogeneous CMPs. Rather than asking programmers to understand architectures and write parallel or low-level device specific versions of their code, we ask programmers to specify and compose algorithms, and we rely on expert programmers to provide efficient parallel implementations of these algorithms, and calls to custom accelerators. This approach seeks to make it possible for non-expert users to take advantage of complex architectures, and to make programs portable across a broad range of architectures.

1. Introduction

Industry is currently exploiting Moore's law by adding processors to homogeneous chip multiprocessors (CMPs). This approach has numerous problems. For example, (1) due to voltage scaling limitations, symmetrically integrating numerous general-purpose high-performance cores is not likely to provide the speedup levels predicted by performance Moore's law; and (2) CMPs require *popular* parallel programming, but despite decades of efforts, producing efficient, scalable parallel programs remains the purview of a few *expert* parallel programmers.

Since voltage scaling has reached its limit, power constraints are likely to prevent activating all cores of a CMP simultaneously, which makes trading cores for extremely power-efficient custom circuits an attractive use of on-chip real-estate. As a result, heterogeneous CMPs composed of general-purpose cores, special-purpose cores (e.g., GPUs), configurable circuits and ASICs may well become the primary approach to performance scaling. To program this hardware, current ISA abstractions are too low level and insufficient. Even programming homogeneous CMPs in this model is tremendously difficult. Programming heterogeneous CMPs with low level ISAs would be a nightmare.

We propose a higher-level *hardware abstraction* that consists of hardware components (e.g., cores and accelerators) implementing common algorithms and a programming model that *explicitly* expresses algorithms. From these algorithm specifications, the compiler or runtime system will automatically map algorithms to customized implementations for cores, multiple cores, or accelerators in a programmer-oblivious way. This hardware abstraction is designed to match what programmers are already doing well—creating complex capable software by composing algorithms. For example, programmers effectively create multiple levels of algorithmic code and data abstractions in the form of functions, methods, classes, class hierarchies, and libraries. When wrapped into *software components*, such codes can be viewed as encapsulating, isolating, and abstracting *algorithms*.

Our approach is *intuitive* because programmers already view programs as a sequence of tasks. The approach is *pragmatic* because it rides the popular software engineering trend of reusing components and libraries. Many current libraries are limited because they impose a fixed coarse level of granularity, but our approach exploits a *hierarchical* expression of the algorithms (e.g., from H264 down to dot-product) so as to impose no specific granularity on the hardware or software. At the same time, the approach delivers *performance portability* because programmers explicitly state which algorithms they use and thus make it possible for the compiler to select the *implementation* best suited for the target hardware: a sequential version, a parallel version, or a call to a hardware accelerator.

While it is unlikely that most programmers can ever write efficient parallel programs, it is likely that, for each algorithm, there exists *a few experts* capable of delivering an efficient parallel version of a given algorithm. We will rely upon a collective development model where components are contributed to free or commercial repositories, in line with recent open or collaborative development practices. Software component developers will be motivated to provide excellent implementations, and hardware vendors will be motivated to accelerate common and widely used algorithms. These components will be *portable*; they will use explicit input and output interfaces, and no internal data will be visible.

In this model, hardware may evolve in two ways: by providing a generic many-accelerator template to easily change and interoperate custom accelerators, and by breeding a broad range of accelerators of different scope and granularity. In this view, most programmers never write parallel programs nor calls hardware accelerators, they only use components containing parallel algorithms or wraps calls to hardware accelerators.

At the moment, we have implemented a prototype version of a runtime system which provides the abstraction layer between a heterogeneous CMP architecture and the software components.

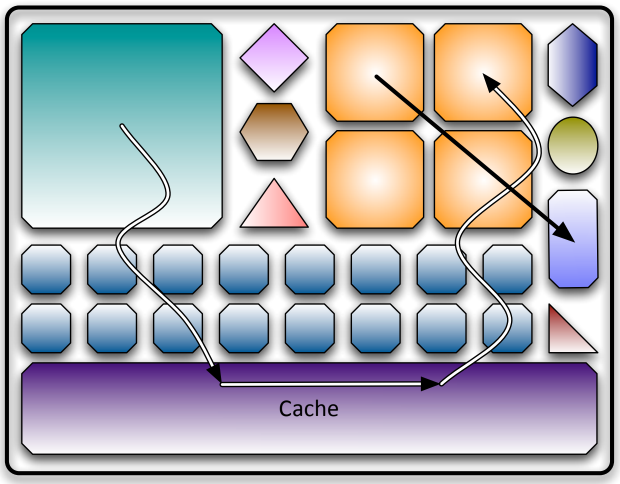

Figure 1. Heterogeneous system-on-chip with various granularity of cores and specialized hardware accelerators.

The purpose of the runtime system is to map the components on hardware (cores and accelerators), and to set up the communication channels between the components corresponding to the components input/output interfaces. For the sake of scalability, the runtime system assumes a distributed-memory model. The prototype implementation is based on knem, a high-performance inter-core communication layer, but we have no preliminary results yet. The remainder of this paper overviews the target hardware, the runtime system, an example program, and then compares our approach with related work.

2. Components for Programming Heterogeneity

2.1 Heterogeneous System-on-Chip

The target of our component-based programming model is a general system-on-chip (SoC) design that integrates various granularities of general-purpose microprocessors, application specific cores, and hardware accelerators. As is typical, the general purpose cores are fully programmable and capable of executing almost any algorithm. The application specific cores, realize a subset of functionality with limited programmability. An image processing core specialized to execute compression and decompression algorithms falls into this category. Hardware accelerators merely realize a certain algorithm such as a JPEG decoder with a minimum required programmability. Figure 1 depicts a high-level view of an SoC design in which the rectangles are general purpose processors of different processing capabilities, and the other shapes are specialized accelerators. Each hardware module executes a given algorithm at a certain level of power consumption and performance. The execution of a program is broken down into several algorithms, each run on one or more of the hardware blocks, and communicating among themselves.

Consider for instance the JPEG algorithm in Figure 2. It is composed of a sequence of three main algorithms: quantization, DCT and run-length encoding. Each algorithm operates on a chunk of 8x8 pixels, so the program breaks down the original image into corresponding 8x8 chunks and then passes them this sequence of three algorithms. In our model, each of these algorithms is a software component running on one or more cores or on a hardware accelerator. The JPEG algorithm itself is a composition of the above three components and thus components form a hierarchy.

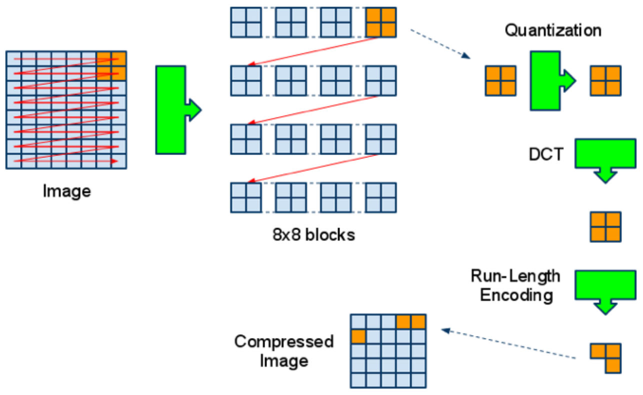

Figure 2. Breakdown of JPEG encoding into three software components: quantization, DCT, and run-length encoding.

The software and hardware components of each algorithm share a common interface, which component implementation is actually used is transparent to the user. The compiler and/or runtime system may directly use a specialized accelerator for some components and/or may choose to compile a software version for a general purpose processor. For an algorithm, there are four types of software components:

1. The software component encapsulating a uniprocessor implementation, which is the default implementation.
2. The software component encapsulating a CMP implementation.
3. The software component encapsulating using a specialized core, such as an image processing core.
4. The software component encapsulating a call to a hardware encoder, which is a custom hardware implementation.

Figure 2 abstractly depicts one component realization of the three steps of the JPEG algorithm: quantization, DCT, and run-length encoding.

2.2 Components as Hardware Abstractions

Our component-based programming model unifies all these three hardware models under the same abstraction enabling a programmer to develop a program as a collection of algorithms, without worrying about the specific hardware blocks these algorithms will be mapped to (GPU, ASIP, FPGA, ASIC). Besides the portability and productivity benefit, it also brings a performance benefit because application may dynamically utilize all the hardware resources available in the SoC. In our model, each component encapsulates either a general purpose core implementing a certain algorithm, or an application specific core, or a hardware accelerator. A specific component can also encapsulate the parallel implementation of the algorithm offering the opportunity of utilizing multiple cores for running the algorithm.

Unified interface of all the four above components is the most important aspect of our component-based approach for heterogeneous SoC programming. That is, the above four components must provide syntactically the same functions with the same function names and hide the data communication between the main application and the hardware module which realizes the JPEG encoder. By adopting the unified interface, the components become swappable or even hot swappable, and may replace each other at runtime.

One requirement is that each hardware module must have its own data transfer mechanism that must be abstracted in the software component. Abstracting the communication mechanism en-

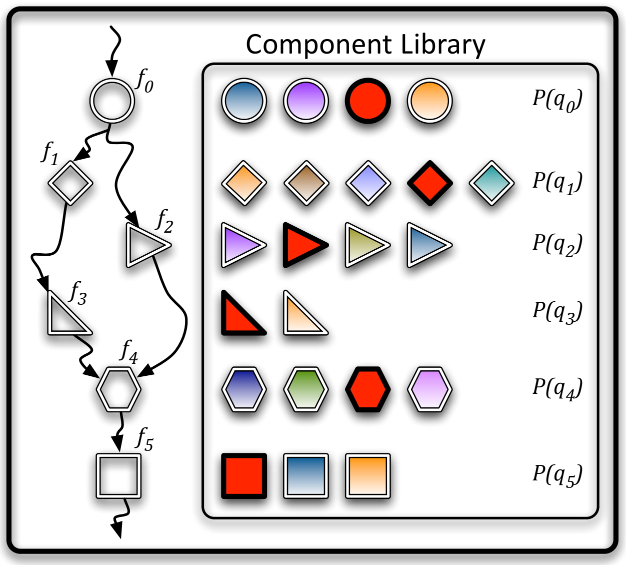

Figure 3. Component view of the program. Each function call is a place holder that can map to any of the available components at run time.

sures that all the components corresponding to the same algorithm can replace each other, even at runtime.

2.3 Programming Heterogeneity

Figure 3 depicts the general program structure that utilizes different unified software components to realize its different functionalities. As shown, the program is divided at the algorithmic level into function calls to the unified component interface. The function calls are place holders that will be mapped to actual algorithm realization depending on which hardware module and its corresponding component is present in the SoC. For example in Figure 3, there are five different software components for realization of f_1. The f_1 function call can be bound to any of those implementations without any change in the application code at compile time or runtime. Each of those implementations deliver a level of performance and consume a certain amount of power for delivering that performance. The power-performance tradeoff of choosing any of the five possible software components for f_1 is denoted by $P(q_1)$. Based on the power, performance, or energy constraints, the compiler or the runtime system decides which software component to pick for the realization of the f_1.

In our model, the programming task is divided between two groups of programmers: (1) component developers and (2) application developers. The former group are the expert group of programmers that know the details of the underlying hardware. The latter group need not be familiar with the details of the hardware and use components to realize their applications independently of the particular component implementations and of how the components interact with the underlying hardware modules. Our model of component-based programming provides the separation of concerns for both programmer groups. The component developers are only concerned with providing a unified component interface for the component encapsulating the hardware, and the application developers are only concerned with the application development and do not deal with the difficulties of interacting with the underlying hardware and transferring data to or from it.

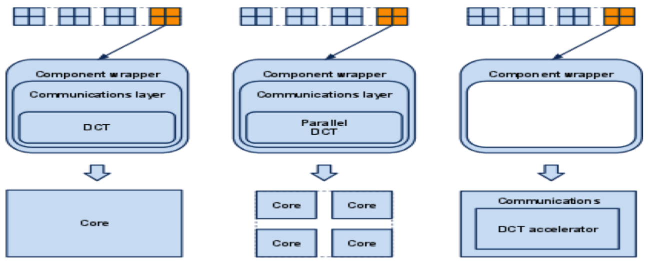

Figure 4. Three components realizing the DCT algorithm over three different hardware modules. Each component comprises a communication layer.

2.4 Distributed Memory Model

To ensure independence and isolation between software components running on different hardware modules, we propose a *logically* distributed-memory model. Each software component only operates on its own local memory without interfering with memory accesses by other software components. The software components communicate with each other using input and output channels. The channels are the only place that two or more software components can transfer data amongst each other. Conceptually, each software component runs on a hardware module with its own memory semantics and then produces an output as the result of running the component. The generated output is sent to another component as its input. Each software component encapsulates the communication semantics in its input and output channels which are an essential part of the component and form the communication layer. This requirement restricts the programming model. For example, it eliminates shared variables between components as a communication mechanism. We believe this restriction will not be too onerous, since application and library developers already structure communications through interfaces, which limits shared variable usage. Figure 4 depicts the communication layer that encapsulates the implementation of the input and output channels between the component implementations of the discrete cosine transformation (DCT) algorithm.

The distributed memory model for the components does not impose any restrictions on the actual memory model used for the intra-component communications. It only defines a unified inter-component communication and sharing semantics. For example, when a component implements an algorithm over multiple cores, the component implementation can use shared memory semantics and use cache coherence and memory consistency to carry out the task. However, after the component has performed its task, the output needs to be sent to another component using the distributed memory model semantics. To efficiently communicate the data between the components, we are developing a library for communication primitives to offer basic input and output channels in the component's communication layer. The communication layer needs to be optimized such that the number of data copies are minimized, while the components can operate interchangeably and in isolation. The next section discusses the details of the runtime environment and implementation.

3. Implementation and the Runtime System

This section describes the syntax and semantics of using components to program heterogeneous system-on-chips through an example that implements the discrete cosine transform (DCT) algorithm. Figure 5 shows two implementations of the DCT algorithm. We extend the C++ programming language with a construct called `component`. The `component` programming construct is sim-

```
component Multicore {
        class InputChannel {
                bool putData(int* srcAddr, int x) {
                        //BEGIN_DEVICE_CODE
                        {
                                inputBuffer = srcAddr;
                                this->x= x;
                        }
                        //END_DEVICE_CODE
                        return true;
                }
        }
        class OutputChannel {
        }
        class Dct  {
                InputChannel inChannel;
                OutputChannel outChannel;

                void compute();
        }
}
```

```
component ImageProcessor {
        class InputChannel {
                bool putData(int* srcAddr, int x) {
                        //BEGIN_DEVICE_CODE
                        {
                                inputBuffer = new int[len];
                                     int i = 0; i < len; ++i)
                                          inputBuffer[i] = srcAddr[i];
                                this->x = x;
                        }
                        //END_DEVICE_CODE
                        return true;
                }
        }
        class OutputChannel {
        }
        class Dct {
                InputChannel inChannel;
                OutputChannel outChannel;

                void compute();
        }
}
```

Figure 5. Two implementations of the DCT algorithm using software components. Both of the components provide exactly the same interface. The communication layer is implemented using the input and output channels.

```
using component ImageProcessor.Dct;
using component Multicore.Dct;

int main() {
        int x = 4, n = 16 * 16;
        int image[n];
        int y, m = n;
        int oImage[m];

        Dct comp;

        s = x * x;
        for (i = 0; i < n; i +=s) {
                comp.inChannel.putData(&image[i*s], x);
                comp.compute();
                comp.outChannel.getData(&oImage[i*s], &y);
        }
}
```

Figure 6. The code running on the master core.

ply a collection of classes that realize the software component. In this example, the communication layer is realized using the `InputChannel` and the `OutputChannel` classes. As depicted, these two classes encapsulate the distributed memory communication semantics. Any communication to the component needs to be done *merely* through these two classes. Since the two components in this example provide exactly the same interface, they are *interchangeable*. That is, the input and output channels are named exactly the same and provide the same methods with unified function signatures. In addition, the `Dct` class is present in both of the components and has a function named `compute()`, which will be called from outside of the component to invoke the component.

Figure 6 shows the main code which calls the DCT component. We have overloaded the semantics of the `using` C++ keyword to inform the compiler that the `Dct` class can either be linked to the implementation deployed in the `ImageProcessor` component or the implementation deployed in the `Multicore` component. Since the two components provide exactly the same interface and communication channels, the body of the `main` function is the same regardless of which of the components used to link to the function calls. The user provides meta-data to the system at compile time that describes the topology of the target SoC system. In our programming model, the compiler can choose statically or dynamically between the implementations. The compiler can use the meta-data to statically link one of the components to the function calls. Alternatively, the compiler does not statically link any of the instantiations or the function calls to any of the components, but it preserves *all* the component implementations in the binary and stores some meta information at each instantiation and function call site indicating which classes and functions can be linked at runtime to the instantiations and calls. When loading the binary, the runtime system the binds the components based on the meta-data and/or the current system load.

3.1 Runtime System

The system can link one time at binary loading time or it can choose multiple times during the execution of the program. In the latter case, there is an active runtime system that monitors the status and utilization of different resources present in the SoC. The runtime could use profiling to monitor the power and performance status of the program. Based on the information gathered regarding the SoC resources status and the program profile, the runtime system could migrate a function from one hardware module to another by hot-swapping the function call and instantiation linkages. For example, another program might have been using the image processor module at the initial loading time of the DCT program, but then later the image processor is free. The runtime system could then swap the function call linkages and start utilizing the image processor. The system might be unplugged from the power source and start to operate using the battery. The runtime system can start shutting down the high-performance high-power hardware modules and start utilizing the low-power components. Hot swapping the components allows the functions to migrate from one hardware module to another in a programmer-oblivious manner.

3.2 Reducing the Communication Overhead using Knem

As mentioned before, our component-based heterogeneous SoC programming model is based on the distributed memory model which allows each of the components to operate in isolation without interfering with other components. As depicted in Figure 6, the distributed programming model requires each component to transfer its input to its logically or physically private memory. In most of the cases, this copying involves transferring data from one process to another process running on a remote hardware module. Such communication involves operating system inter-process communication primitives. That is, the data needs to be copied to shared pipe and then from the shared pipe to the local buffer of the remote process. It means each data transfer involves two copies, which is too costly, when data is large. To reduce this overhead, we have developed a inter-component communication library based on KNEM [3]. The KNEM library is a linux kernel module developed for MPI communication on chip multiprocessors. The KNEM kernel module, allocates a region of memory in the kernel space that can be accessed in the user space using the KNEM ABI. Using KENM, the data only needs to be copied once to the KNEM memory region. Then, the information of the KNEM region can be passed to the remote process. The remote process does not need to copy the data and can start operating on it once it has the appropriate handle. Given the latency of the Linux operating system, when the data is in the order of dozens of kilobytes, the benefit of one-copy data transfer compensates for the overhead of trapping to the OS. The KNEM library alleviates half the copies on large data transfers for our component-based and distributed-memory heterogeneous SoC programming model.

4. Related Work

The notion of partitioning programs into blocks which are then individually mapped to hardware resources pervades much of the research on parallelization. Cilk [2] and Thread Building Blocks (TBB) [8] present programs as a set of code blocks which can be dynamically split and mapped to cores. Charm++ goes one step further and applies that same notion to distributed-memory parallelization by strictly encapsulating code blocks into independent modules that communicate over explicit input/output ports, similar to components [5].

Merge is an environment which bears some resemblance to our approach by combining the notion of independent code parts which can be indifferently mapped to different hardware tiles [6]. However, it is solely focused on MapReduce-type of parallelization. As in our approach, PetaBricks advocates an algorithmic-centric view of programming, akin to a generalized used of libraries [1]. However it only deals with shared-memory parallelization and relies on the programmer to directly implement, within the program itself, the different versions of each algorithm, instead of relying on an external repository of components sharing a common and compatible interface. COMPASS does not advocate similar software-based methods for realizing the compatibility among code parts, but it does promote the notion of community-based programming, in the spirit of the repositories where expert users would contribute components [9].

The notion of components itself is popular in software engineering, such as for the Java libraries, Java Beans [4], and Microsoft .Net [7] components. The software goal goal for components is to improve programming productivity by implementing isolation and explicit communications to facilitate reuse and ensure the isolation of concerns. We seek to extend the benefits of components to include performance as well.

5. Conclusions

We have proposed a programming approach based on the decomposition of programs into independent software components, each corresponding to an algorithm. A repository of components contains a range of implementations of such algorithms, from classic sequential programs, to parallel programs and wrappers of hardware accelerators. Because components share a common interface, they can be seamlessly interchanged. As a result, these components empower the non-expert user to seamlessly take advantage of the parallelization and acceleration features of a complex heterogeneous CMP. Our implementation is focused towards a distributed-memory implementation, typical of system-on-chips and requires no cache coherence. We have presented a first version of the runtime system that maps the different components to the cores and/or accelerators and that connects them for data transfers. This approach provides a new distribution of roles between the application developer who uses components, and the expert developer who creates versions of algorithms for different types of architectures.

References

[1] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. Petabricks: a language and compiler for algorithmic choice. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 38–49, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1. doi: http://doi.acm.org/10.1145/1542476.1542481. URL `http://doi.acm.org/10.1145/1542476.1542481`.

[2] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming*, 1995. URL `citeseer.ist.psu.edu/blumofe95cilk.html`.

[3] D. Buntinas, B. Goglin, D. Goodell, G. Mercier, and S. Moreaud. Cache-efficient, intranode, large-message mpi communication with mpich2-nemesis. In *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP '09, pages 462–469, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3802-0. doi: http://dx.doi.org/10.1109/ICPP.2009.22. URL `http://dx.doi.org/10.1109/ICPP.2009.22`.

[4] L. G. DeMichiel. *Enterprise JavaBeansTM Specification, Version 2.1*. Sun Microsystems, Nov. 2003.

[5] L. V. Kale and S. Krishnan. CHARM++ : A Portable Concurrent Object-Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 91–108. ACM Press, Sept. 1993. URL `citeseer.ist.psu.edu/95307.html`.

[6] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng. Merge: a programming model for heterogeneous multi-core systems. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, pages 287–296, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-958-6. doi: http://doi.acm.org/10.1145/1346281.1346318. URL `http://doi.acm.org/10.1145/1346281.1346318`.

[7] A. Rasche and A. Polze. Configuration and dynamic reconfiguration of component-based applications with microsoft .net. In *ISORC '03: Proceedings of the Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'03)*, page 164, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1928-8.

[8] J. Reinders. *Intel threading building blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007. ISBN 9780596514808.

[9] S. Sethumadhavan, N. Arora, R. B. Ganapathi, D. John, and G. E. Kaiser. Compass: A community-driven parallelization advisor for sequential software. In *Proceedings of the 2009 ICSE Workshop on Multicore Software Engineering*, IWMSE '09, pages 41–48, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3718-4. doi: http://dx.doi.org/10.1109/IWMSE.2009.5071382. URL `http://dx.doi.org/10.1109/IWMSE.2009.5071382`.