國立交通大學

多媒體工程研究所

碩 士 論 文

支援多核心架構之程式轉換技術

Program Transformation for Multi-core Architecture

研 究 生：王勝保

指導教授：陳俊穎　教授

中 華 民 國　九 十 七 年 八 月

支援多核心架構之程式轉換技術
Program Transformation for Multi-core Architecture

研 究 生：王勝保　　Student：Sheng-Pao Wang

指導教授：陳俊穎　　Advisor：Jing-Ying Chen

國 立 交 通 大 學

多 媒 體 工 程 研 究 所

碩 士 論 文

A Thesis

Submitted to Institute of Multimedia Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

Aug 2008

Hsinchu, Taiwan, Republic of China

中華民國九十七年八月

支援多核心架構之程式轉換技術

學生：王勝保　　指導教授：陳俊穎 博士

國立交通大學多媒體工程研究所

摘　　要

發展嵌入式系統時須考量不同軟硬體平台的特性及各類系統資源的有效運用。在未來以多核心架構為基礎的嵌入式系統成為主流後，如何針對不同平台特性設計系統提升整體效能，並同時降低發展成本變得更加重要。針對這個問題，本論文提出使用程式轉換的技術，使得開發者可專心發展應用程式，不須考慮過多軟硬體平台的細節，並能仰賴各種程式轉換工具針對不同的軟硬體特性產生適用的程式碼。為了驗證此方法的可行性，我們以 Java 語言為基礎，在不同嵌入式平台，包括一個雙核心架構的平台上作試驗，並得到實驗結果的支持。

Program Transformation for Multi-core Architecture

Student : Sheng-Pao Wang Advisors : Dr. Jing-Ying Chen

Institute of Multimedia Engineering
National Chiao Tung University

ABSTRACT

Embedded systems are characterized by their scarce computing resources and heterogeneous hardware-software configurations. With multi-core architecture entering the embedded systems market, developing efficient software applications, and delivering them timely, becomes even more challenging. One main obstacle to embedded software development is to tune applications for different system configurations in order to maximize system performance in terms of execution speed, memory, energy consumption, and so on. Often, a particular software design that performs well in one configuration may work miserably in another. The objective of this thesis is to investigate the use of program transformation techniques as a solution to this problem. The idea is to have a framework where developers can concentrate on developing applications without devoting excessive effort on low-level hardware and system software details, and rely on different program transformation schemes to produce programs for specific platforms. With the framework, we have experimented with various Java benchmarks on different platforms, including an embedded platform containing two cores. The result shows that program transformation can improve performance significantly with considerably less development and tuning effort.

誌　　謝

對於學位論文的完成，首先必須感謝我的指導教授陳俊穎老師，在研究所的兩年求學生涯中，總給予我詳盡與切要的指導。指引我正確的研究方向，對於解決問題的方法和研究態度上，也使我獲益良多；同時特別感謝口試委員楊武教授與黃慶育教授在百忙之中給予許多寶貴的指導與建議，使得論文的內容更加完備。

此外，感謝研究室一起奮鬥的伙伴們，亦超、宜涼，以及登揚諸位學長，以及坤定、仁傑兩位學弟，在研究進行時給與我許多的支持與鼓勵，並伴我度過這兩年的研究生涯。

最後，由衷地感謝我最親愛的家人們，正因他們全力支持與包容，才能促使我順利的完成學業，願將這份成果共享於我的家人。

王勝保 謹誌 2008 年 8 月

於交通大學協同合作實驗室

Table of Contents

List of Figures and Tables

Chapter 1. Introduction

Embedded systems range from tiny sensors that possess limited computing power but consume little energy, to battery-powered hand-held devices supporting multiple software applications interacting with users via various types of user interfaces, to set-top boxes with increasing computing capability and more demanding energy requirement. Software development plays an increasingly important role in any embedded system projects. To cope with heterogeneous hardware/software configurations each with different budget in terms of computing power, memory usage, energy consumption, and the types of peripherals, software engineers need to spend a great deal of effort on configuration-specific customization, hoping to tune the system for optimum performance within the given budget. Such relatively "mundane," time-consuming task, compared to the development of the applications for end users, is becoming more burdensome when systems become more powerful capable of hosting multiple applications simultaneously. The problem is further intensified by the fact that multi-core architecture, developed mostly for higher-end, desktop systems previously, is also gradually entering the embedded systems segment. The implication of such a shift is quite significant for software engineers, because developing concurrent, multi-threaded programs running on multiple processors simultaneously has always been challenging when compared to single-threaded applications. As if it is not enough, concurrent programming is more burdensome if the multiple processes are in fact heterogeneous each with special capabilities (e.g. signal processing, graphics rendering, networking, etc.). When the universal time-to-market factor dominates the entire development project, it is not surprising to see that many delivered embedded systems operate in a sub-optimal mode.

Advanced CAD tools have always played a crucial role for hardware and other engineering disciplines, to help engineers analyze, validate, and improve their designs continuously and rapidly. Development of CAD tools for software engineering, although long thought to be desirable and critical, has not kept up with the pace of advancement of software development technologies. Although this thesis is not the place to investigate the main causes of this issue, it is clear that for in the field such as embedded systems where innovations emerge rapidly, sufficient CAD support will always lag behind. The question is how to develop CAD tools that is sufficiently useful, fast enough, while keeping the development code low so that the return of investment is justifiable.

It should be noted that the fundamental tool-chains and associated development environment are always required for any embedded system project. What we mean by CAD support are those “additional” tools that, among other things, help engineers visualize and analyze the architecture, design, and implementation of the system under development. Without CAD support, engineers mostly have to rely on manual programming and tuning of the system, based on previously obtained execution history or performance measures.

Nevertheless, the scope of CAD support in general or for embedded system development in particular, is still too large. What we are interested in are strategies and mechanisms that help engineers derive applications for different system configurations easily without having to modify these applications thoroughly. To be more specific, there are some questions we want to address:

- How to adjust an application if the memory budget is low. What are the available maneuvers that can be tried?
- How to partition an application into modules that can be deployed on different cores, taking into account the necessary interface and communication changes between modules.
- For an application written with single-thread execution in mind, how to adjust it in case the underlying system support multi-threads.
- How to quickly obtain empirical feedback in order to understand the impact of various optimizations, their improvement or mismatch between anticipated result and actual result.

In this thesis we focus on program transformation techniques, and pay particular attention to their usefulness for embedded software development. Such a source-level transformation approach is highly portable from one processor to another and is complementary to existing platform-specific, back-end optimizations. With sufficiently powerful program transformation tools, the task of developing higher-level, end-user applications can be decoupled from low-level system-specific optimization more desirably. For example, instead of injecting various compiler flags and machine dependent code into the main code base, the engineers may instruct the program transformer with various rules or heuristics to produce final programs geared towards specific target platforms. Moreover, by putting less effort into compiler development, the transformational approach help reducing

the time to market of the embedded system product while giving higher performance. Even though it is arguable that the generated programs are still not optimum when compared with those hand-crafted equivalents – just like the general perception that compiler-generated code cannot compete with hand-crafted assembler code – we argue that software engineers gain substantial advantages when they use program synthesizers properly. For example, it would be faster to explore different alternatives and make better design decisions without indulging prematurely into hand crafting tasks.

The rest of this thesis is organized as follow: in chapter 2 we will survey related areas in compiler techniques and program transformation systems for embedded systems. In chapter 3 we will outline the design objective, considerations, and the architecture of our program transformation framework. In chapter 4 we will describe and discuss some performance studies using the transformation framework, as well as some experiments with an experimental embedded system board. In chapter 6 we will conclude the thesis.

Chapter 2. Background

Embedded systems are essentially a highly dynamic field because of the fast pace of innovations and development in hardware, operating systems, and application development technologies. With the tremendous commercial potential at stake, as well as relatively smaller investment companies require to join the game, it is not difficult to see the embedded systems market is filled with a huge and diverse array of hardware devices and associated system software platforms. This suggests that the dimensions of design space for embedded systems also increase dramatically. One would have to consider target-system-specific transformations, such as memory optimization requiring target architecture-dependent loop transformations, optimized word length selection, and process restructuring for fine-grain load distribution, etc. As stated in [Ernst], "the problem is worse here than with parallel compilers because of architecture specialization."

Research interests in optimization techniques for embedded systems have grown in recent years, in an attempt to understand the various optimization strategies applicable to the design of embedded systems in terms of time-space performance and/or power consumption. For example, some conferences and workshops such as LCTES and ODES are devoted to such goal. Compiler and optimization research for embedded systems shares the same foundation with general compiler techniques, but has specific challenges to address:

- **Energy awareness**. Many approaches address the energy consumption issues that are crucial for embedded systems (e.g. [Lambrechts], [VanderAa03], [VanderAa05]).
- **Memory hierarchy**. Some compiler techniques explore the characteristics of memory hierarchies that differ from one embedded system from another ([Chen], [Grewal], [Ozturk], [Sanghai07]).
- **Transformation and code generation**. There are also efforts that propose code generation and transformation techniques for embedded systems. In addition, there are also efforts concerned with specific problem domains such as algorithms for FFT or other multimedia applications, and develop specific code generation methods for embedded systems. Examples include [Ali], [Alur], [Burgaard], [Cheng], [Franke], [Hong].

- **Multi-core architecture**. Work in this category concerns the parallelization aspect for multi-core embedded systems or more general multi-processor architecture ([Dupre], [Perkins], [Sanghai05]), and is closely related to traditional parallelizing compiler research..

- **Profiling-based optimization**. Work in this category proposes methods that guide compiler with profiling information, possibly obtained from actual execution of automatically generated programs. ([Cavazos], [Peri], [Zhao05], [Suresh]).

- **Framework and methods**. There are also research efforts proposing general framework and methods that can are applicable for embedded systems. ([Aarts], [Barat], [Fulton], [Pan], [Vachharajani], [Zhao03]).

The references presented above are just samples from the vast literature in compiler research. Furthermore, the categories above are not mutually disjoint, since it is often the case that a research effort will consider multiple aspects simultaneously. In what follows we focus on research and development relevant to the area of compiler techniques, or more specifically program analysis and transformation systems for multi-core embedded systems, without probing further into areas such as energy consumption or more hardware-oriented research.

For efforts related to program transformation, [Franke] investigates source-level transformation for embedded systems, which incorporates a probabilistic feedback-driven search for proper transformation sequences. Specifically, it combines a simple random search for space exploration and a focused search based on a machine learning approach, in order to help reducing the extremely huge search space. [Lee] investigates the case of dual instruction set processors that are increasingly popular for embedded systems. Typically, programs compiled with a reduced instruction set (16 bits/instruction) have smaller code size but run slower, but run faster with larger code size when compiled with a full instruction set (32 bits/instruction). Thus [Lee] first compiles a program with the reduced instruction set first, and then analyses and selects a set of basic blocks of the program and transforms them using the full instruction set that gives the maximum performance gain while maintaining the code size under a given upper bound.

When the development of the hardware is also part of an embedded system project, the matter becomes more involved. [Ernst] provides an overview of research in hardware/software co-design of embedded systems. [Bennett] proposes a framework that

combined automated code transformation and ISE generators to explore the potential benefits of such a combination. Although the design space is even larger in the hardware/software co-design context, what was discussed previously is still useful. Still, there are a number of points worth pointing out:

- An integrated and coherent co-design system should capture the complete design specification, including hardware and software models, and support design space exploration with optimization based on this specification
- Synchronization and integration of hardware and software design becomes an issue.
- The boundary between hardware and software is interesting. Although this aspect resembles the boundary between programs written in high-level languages and those using assembly languages (to boost performance for a critical part), the hardware/software boundary is “permanent” and thus requires much rigid analysis and careful decisions.

General-purpose source code transformation frameworks are also relevant in our study. For example, [Cordy] presents the TXL system that supports all aspects of parsing, pattern matching, transformation rules, application strategies and unparsing within a specially designed language with no dependence on other tools or technologies. [Bravenboer] is another framework, called Stratego/XT, that is essentially a collection of reusable components and tools for the development of transformation systems, where transformation components are implemented using the Stratego language that provides rewrite rules for expressing basic transformations, programmable rewriting strategies for controlling the application of rules, concrete syntax for expressing the patterns of rules in the syntax of the object language, and dynamic rewrite rules for expressing context-sensitive transformations.

Another important category of compiler research for embedded systems concerns parallelization. For multi-core architecture this usually amounts to the partition of a program into parts each may sit on different cores. [Suresh] provides both compiler-based and simulation-based loop analyzers that profile an application, and a loop analysis toolset to support hardware software partitioning. These tools are used to identify core fragments of programs of many benchmarks that are executed most frequently. However, these core fragments are factored out into hardware in a manual way. [Kim] presents a source code analysis technique that, although not targeting embedded systems per se, is still relevant to

parallelization and program transformation. The technique attempts to extract so-called pre-execution code from an ordinary program such that the pre-execution code can be placed as another helper thread that runs “in spare hardware contexts ahead of the main computation to trigger long-latency memory operations early, hence absorbing their latency on behalf of the main computation that can be issued prior to the main program.”

When more general compiler techniques are concerned, the space for exploration includes many dimensions. A particularly important dimension is the kinds of optimizations that can be performed. This dimension is profound due to the ever increasing complexity of hardware architecture. There are also enormous research efforts on applying existing, general optimizations to embedded systems. An even more challenging dimension regards the potential interferences among optimizations. It is well known that the order of optimizations can affect the quality of the final result; some optimizations may enable or disable future optimizations. [Zhao03] investigated the impact of optimizations to embedded systems. [Franke] mentioned above also discusses its program transformation framework for embedded systems from this optimization space exploration perspective. The iterative compiler approach (e.g. [Aarts]) addresses this by performing optimizations in different ways, and observe the performance characteristics of the actual generated code rather than relying on heuristics or abstract performance models. [Triantafyllis] proposes a more elaborated framework for exploring the space of optimizations. In this framework, a compiler optimizes each code segment with a variety of optimization configurations and examines the code after optimization to choose the best version produced. Because this finer-grained iterative compiler approach results in larger search space, the framework also proposes methods to reduce the search space. [Pan] builds a feedback-directed optimization orchestration algorithm which searches for the combination of optimization techniques that achieves the best program performance. The algorithm attempts to successively identify and removes harmful optimizations, measured through a series of program executions, with the goal to reduce the number of compilations while maintain the quality of the generated code.

Because our transformation framework is geared towards Java programs, it is also worth considering related work for embedded Java. The Java Platform, Micro Edition (J2ME) is a set of technologies and specifications developed for small devices like pagers, mobile phones, and set-top boxes. J2ME uses smaller-footprint subsets of Java SE components, such as smaller virtual machines and leaner APIs, and defines a number of APIs that are specifically

targeted at consumer and embedded devices. It is proposed to enable users, service providers, and device manufacturers to take advantage of a rich portfolio of application content that can be delivered to the user's device on demand, by wired or wireless connections.

Although JVMs can be implemented using straightforward interpreters, the most popular approach to improving JVM performance is to replace or augment the interpreter with a just-in-time (JIT) compiler, which transforms the bytecode into machine code that can be executed by the host machine directly. Depending on the level of optimization applied, the translated machine code may even approximate the speed of equivalent programs written in C.

However, the JIT compilation phase may be quite complicated, also depending on the type of optimizations performed, and require substantial memory and processing time. In addition, especially for embedded systems, the impact of this in terms of user experience can be very significant, particularly at application start-up, as a device appears unresponsive for a long period of time. It becomes obvious that compiling all bytecode into native code may incur too much overhead, especially statistically speaking a large portion of the bytecode is executed vary rarely or even not executed at all.

Accordingly, some researchers try to design lightweight and efficient JIT compilers. For example, in [Tabatabai] a JIT compiler for the Intel IA32 architecture is proposed that generates native IA32 instructions directly from the byte codes, in a single pass. Other than a control-flow graph used for register allocation, the JIT does not generate an explicit intermediate representation. Rather, it uses the byte codes themselves to represent expressions and maintains additional structures that are managed on-the-fly. This is in contrast to other Java JIT implementations which transform byte codes to an explicit intermediate representation. Another example described in [Shudo] presents cost-effective code generation and optimization methods by means of template connecting. The code generator basically connects pre-fabricated templates of native code corresponding to internal instructions. In addition to the technique, stack caching [Ertl] was implemented in the compiler and the technique makes use of multiple registers over templates.

Therefore, most JIT compilers are selective about the bytecode to compile, often on a method basis. That is, they collect the frequency of each loaded method and only compile those methods that are (expected to be) frequent. Furthermore, because the more

optimizations to apply, the more memory and processing time are needed, many JIT compilers have multiple levels of optimizations each with different memory and processing time requirements, so that different optimizations can be applied to different methods based on their relative frequencies.

Another way of addressing the issue of compilation overhead is through the so-called dynamic adaptive compilers (DAC), sometimes also referred to as mix-mode interpreters. In a DAC, bytecodes are initially executed by interpretation while software profiles the code and determines key code sections to be compiled. Some variations avoid the interpretation all together and convert the bytecode into native code immediately using inexpensive translation. Once the key code sections, mostly methods, that are identified as hot, they are compiled using more advanced optimization options. Furthermore, the available levels of optimizations may be more than one, so that only extremely hot sections receive extensive analysis and optimizations. The popular Jikes RVM (research virtual machine) employs such an MMI approach.

Another category of JVM optimization that is more relevant to our study is the so-called ahead-of-time (AOT) compilation. As the name suggests, if the Java programs and/or bytecode are known prior to their execution, we can employ traditional compiler steps by translating them into native code in advance. Extensive program analysis and optimizations can therefore be applied without limitation. We believe AOT compilation is a very important technique for embedded systems because there are often core applications that should be bundled with a given embedded system. Note that if the embedded Java system is required to download and execute bytecode dynamically, JIT compiler is still essential.

Optimized ahead-of-time compilation attempts to produce code having size and speed comparable to code written in C/C++, while remaining compatible with the Java world, allowing for the mixing and matching of code according to individual system requirements. Some AOT compilers such as GCJ translate Java programs into native code (through the common GCC backend), while others may translate Java into C and rely on C compiler to perform sophisticated optimizations. Note that AOT compilation does not conflict with JIT compilation. In fact, many AOT compilation frameworks still require JVMs to process and execute bytecode if it is allowed to execute Java applications containing both natively compiled part and bytecode part.

AOT compilers are usually not for dynamically downloaded classes. It is nevertheless possible to invoke existing JIT compilers on bytecode to obtain and cache the compiled machine “ahead of time,” thereby reducing the time for class loading and optimization substantially. This simpler approach to AOT compilation comes almost free because it makes use of the JIT compiler that already exists. CVM actually provides such facility that allows engineers to customize additional classes for AOT compilation and to include them in the final binary footprint. [Hong] also extends from the idea and that proposes a “client-side” compilation during run time for downloaded classes. The idea works because in certain embedded applications such as set-top boxes where applications may be executed many times after they are downloaded, hence the time saved for class loading and optimizations can be significant.

Chapter 3. The Program Transformation Framework

Program transformation is a powerful technique that has been used to support various kinds of software engineering activities. In fact, refactoring – the process of restructuring code with the purpose of making it easier to understand and maintain without changing its observable behavior – is strongly coupled with program transformation. Nowadays, the use of refactoring to increase quality is considered a very important development practice. For instance, Extreme Programming, an agile approach to software development, includes refactoring as a standard activity to improve software design continuously.

Unlike compilers, many program transformation tools are not language specific, being able to transform programs from an arbitrary source language to an arbitrary destination language. In general, a program transformation tool requires two kinds of user: the transformation engineer, who configures the tool (encodes the transformations) and the programmer, who uses the tool for software development (applies the transformations). While this provides necessary flexibility, it complicates the use of the tools. As a result, making use of program transformation in practical, large-scale projects to improve productivity is not possible without sufficient tool support.

3.1. Objective and Considerations

Our goal is develop a program transformation framework that can benefit software development in general and the development of multi-core embedded systems in particular. It is beneficial to envision a full blown, fully integrated CAD environment that can assist developers in all aspects, even though our actual goal here is far less ambitious. By identifying the long-term directions, we hope to design an extensible framework that has core assets sufficient for short-term needs and can be extended further for new, possibly much more sophisticated features in the future. Hence below we proceed with the discussion of a comprehensive program transformation system, its ideal features, requirements, and potential issues.

A complete program transformation system should capture the complete design specification of an embedded system, support design space exploration with optimization based on this specification, and even to deal with different aspects of developing and

finalizing the embedded system. For example, some embedded systems may be equipped with primitive operating system without real-time facilities. Generating programs for them will be quite different from generating for other systems with real-time support. Yet, whenever possible, one should avoid developing the same application twice just for these two classes of embedded systems.

Without relying unrealistically on compilers to extract concurrency from an arbitrary single-threaded program and to re-shuffle it to fit different target platforms, a more cost-effective approach is to only work on programs that are written with certain styles, possibly with additional annotations, to help generators generate target-specific programs more suitably, or at least suggest potential improvements one can attempt to take better advantage of the generators' capability.

The goal of the generator can be further generalized to cover hardware/software co-design or even co-synthesis. That is, within given hardware and software requirements and application domains, it is possible to investigate different system-wise designs. To pursue this goal, however, more detailed hardware model and software model should be used. Because of the diversity in possible hardware and software models, developing a comprehensive environment supporting all of them is probably too much.

In addition to the general objective and considerations mentioned above, we also have a more project-specific objective and associated considerations. One of the major test beds of our transformation framework is the PAC (Parallel Architecture Core) platform. Figure 1 depicts the overall architecture of the PAC platform, which is designed to provide a platform for the next-generation mobile devices such as smart phone, PDA, and portable media players. The PAC platform features a dual-core architecture, where the MPU core is designed to execute system and application programs, while the DSP core is suitable to execute complex computation programs, such as multi-media codes.

One of the major milestones of the PAC project is the development of a high-performance compiler and related development environment for the DSP core. Therefore, to help unveiling the potential power of the PAC platform, our framework should be used to help offloading some of the computation tasks from the MPU core to the DSP code.

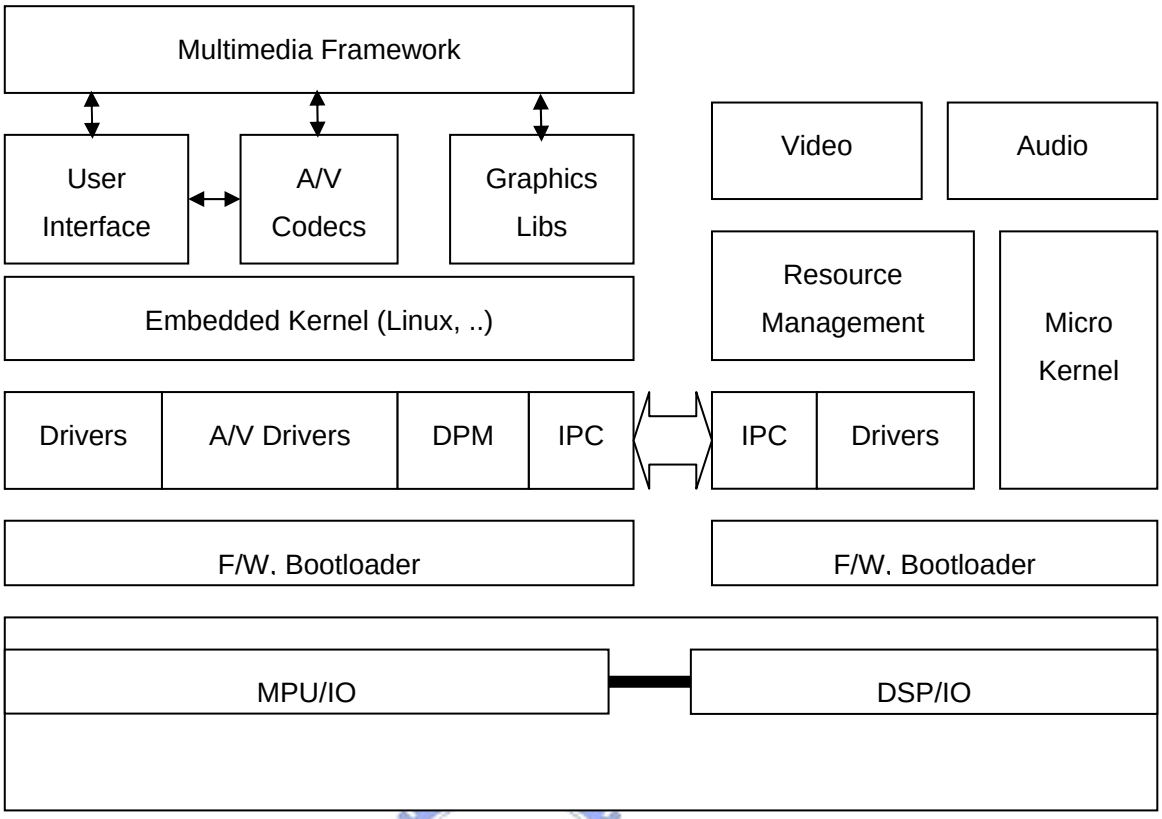


Figure 1. Portable PAC SoC Platform

A critical concern, however, is the fact that the benefit of such labor division may be overshadowed by the incurred initialization and communication overhead. Furthermore, the different computing models these two cores are based on can introduce additional overhead. For example, when the MPU is running Java-based application while the DSP is based more on C-style procedural execution, there will be additional run-time adaptation to cope with object instantiations, structural traversals, as well as object-oriented inheritance mechanisms. Clearly, the transformation framework should provide engineers with means to quickly assess the potential gains or overhead of various load sharing schemes. It is also desirable that the framework can analyze the initial program and suggests to the engineers promising candidates or potential bottlenecks.

3.2. System Requirements

Based on the considerations given previously, we present a program transformation framework with the following general objectives:

- It is able to explore large design space without particular restriction on target

platforms. For different platforms, it is also able to recognize its strength and limits to better harness their potential power.

- It is extensible in many dimensions: new target platforms, new generation and optimization schemes, new source and target languages, and so on.
- It is relatively easy for engineers to experiment different exploration and generation heuristics without writing intensive algorithms using the development programming language.
- It provides sufficient facilities that help engineers obtain execution or simulation feedbacks in a timely fashion; this requires the framework to streamline the transformation process seamlessly.

Under the general requirements, our near-term goal is to mainly focus on Java as the base language, with target platforms including X86 and ARM architecture, where the later includes both emulated ARM machine as well as the PAC platform. Below is a list of specific requirements for the near-term system

- It generates platform-specific C code from input Java programs, partly or whole. When only part of the Java programs are generated, the communication between the Java run-time and the C counterpart is through JNI (or KNI when the CLDC KVM is used).
- It provides profiling and performance evaluation facilities by augmenting the JVM with proper instrumentation mechanisms.
- It allows developers to experiment different exploration strategies easily, possibly using a rule-based language. Of course, when innovative exploration schemes are sought that are beyond what existing facilities are capable of describing, the framework is still extensible, in terms of new analysis algorithms and search procedures in Java (the implementation language of the framework), so that the user has a systematic way to adjust the system for his/her investigation.
- When exploiting the multi-core architecture of the PAC platform, the framework supports different communication schemes between the MPU and the DSP cores. In particular, direct communication or inter-process communication via micro kernels (and IO interrupts) should be considered.

It is also worth mentioning the long-term objective of our framework, which also indicates the aspects that are not addressed in the short-term objectives.

- It supports multiple base languages. This is to recognize that most embedded systems rely on C or C++ as the primary development languages. This is also reflected by most research and development projects for embedded systems. As also mentioned in Section 2, researchers also recognize the importance of generating target-specific programs based on platform-neutral programs.

- It supports useful visual tools. There are different kinds of visual tools that can help increasing the productivity and quality of the generation process, hence the overall development process. For example, the user may describe an abstract machine model with multiple cores and describe each core with specific characteristics. This can be combined with the visualization of the input programs and generated programs (as modules) to show the result of transformation or even the simulation or execution results.

- It is self-applicable. This is an important long-term goal of our framework. Because the rule-based language itself is also a powerful language, it also makes sense to process the input rules and to generate optimized Java programs to speed up the transformation process.

- It supports optimization-space exploration. Because there are potentially unlimited optimizations conceivable for various program styles or other conditions, orchestrate these optimizations suitably to achieve better results may be challenging. The framework should permit semi-automatic to fully automatic means for engineers to blend different optimizations or program refactoring.

3.3. Architecture Overview

In this section we outline the overall architecture of our framework. Figure 2 shows the role of the transformation framework in the overall development and execution environment of an embedded system.

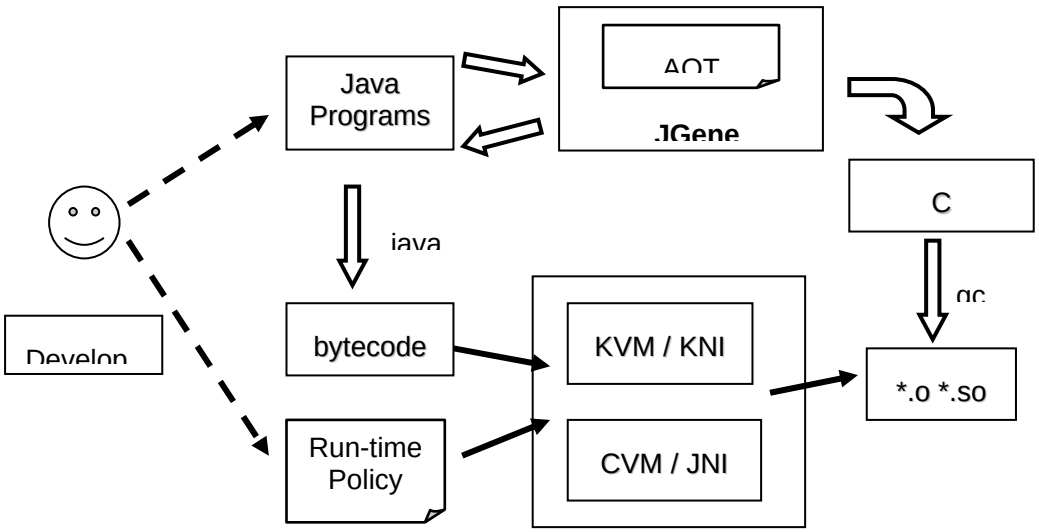


Figure 2. The Overall Program Transformation Framework

In this embedded system development environment, the developer is expected to write pure Java programs that are intrinsic to the problem at hand, and then use the transformation subsystem to selectively factor out part of the Java program into platform-specific C programs. For different target platforms and different design constraints, the engineer needs to create specific polices or rules to instruct the transformer, or even additional platform-specific modules or utilities.

The JGene subsystem shown in Figure 2 is the core of the overall framework. It is an extensible object-oriented framework for Java program transformation. One notable goal of JGene is that the engineers could write embedded applications in pure Java and test their functionalities immediately and translate critical parts into more efficient C/C++ programs. This has great potential in reducing the turnaround time for (embedded) software development.

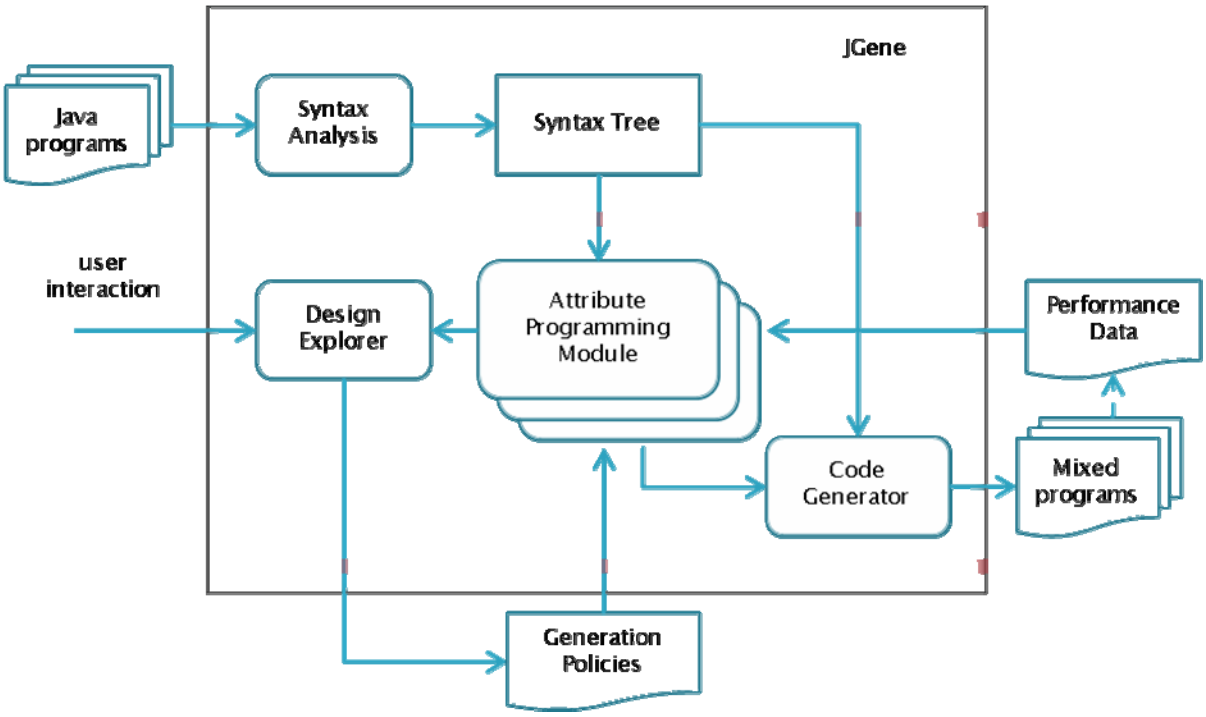


Figure 3. The Overview of the architecture of the JGene core

Figure 3 above shows in more detail the internal design of JGene. In addition to the usual abstract syntax tree (AST) module, JGene places an *attribute programming* module at the core. The attribute programming module basically provides the developers extension mechanisms to define new types of attributes for Java ASTs, as well as the propagation rules among them. Although the basic idea is almost the same as attribute grammars, there are some differences. For example, many transformation tools, including back-end code generation components that employ the attribute grammar, let developers define attributes and auxiliary subroutines alongside the syntax definition (and rely on syntax-directed translation to produce the final translator). Although compact and straightforward, this approach introduces several problems. First, the resulting system tends to be large, cluttered with each other and the syntax definition (also because the grammar itself is usually very large). More importantly, this form of programming is not extensible, as new transformation scheme will have to be written with substantial duplication.

Another limitation for typical transformation tools is the lack of flexible support for *controlling* the propagation of attributes as well as *reasoning* with the interrelations with attributes. Either these aspects are left as the developers' responsibility, or they restrict what

the tools can support.

As an object-oriented framework, JGene provides a general model for defining and programming with attributed defined by the developer. The idea is illustrated in Figure 4. The attribute programming module provides core classes represents attributes: Attribute and its subclass Choice. While most attributes are computed from ASTs and other contextual information, Choice represents user's decisions in some aspects defined by the user. The example in Figure 4 indicates that, for some AST elements, the user may choose to generate them into pure C program, JNI-compliant programs, or pure Java.

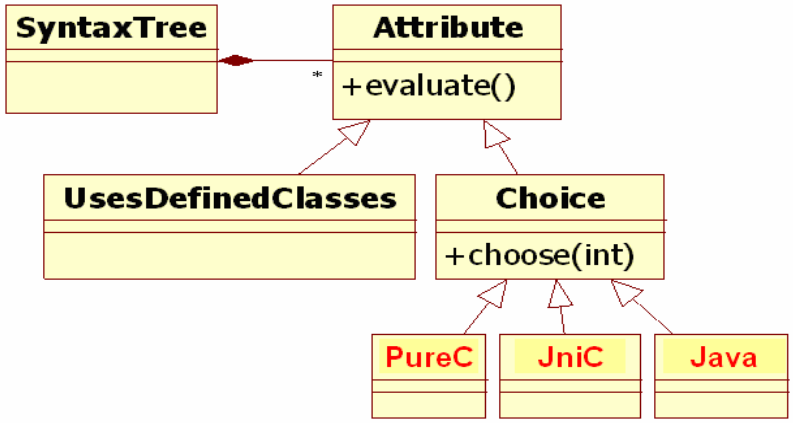


Figure 4. JGene program attributes

In fact, each AST element may be generated into different types according to user-defined attributes and some binding rules. We list a few important sample attributes below:

- ***PureC***: the element is to be translated into (ANSI) C; the value may be true, false, or undecided.
- ***JniC***: the element is to be translated into JNI-compliant C; the value may be true, false, or not-decided.
- ***Java***: the element remains in pure Java; the value may be true, false, or undecided.
- ***UsesDefinedClassesOnly (UDCO)***: the element does not use system or external library classes; the value may be true or false.
- ***MethodCallers***: the sets of method invocations (which are AST elements) calling this element, attribute value may be null or a set of elements.

3.4. Design Space Exploration

After attributes are defined, the developer in turn specifies the dependencies among these attributes. Some attributes require input from the user; other attributes may depend on these attributes. Moreover, because some choices may imply or invalidate other choices, the attribute programming module in effect becomes a simple, but convenient constraint programming tool.

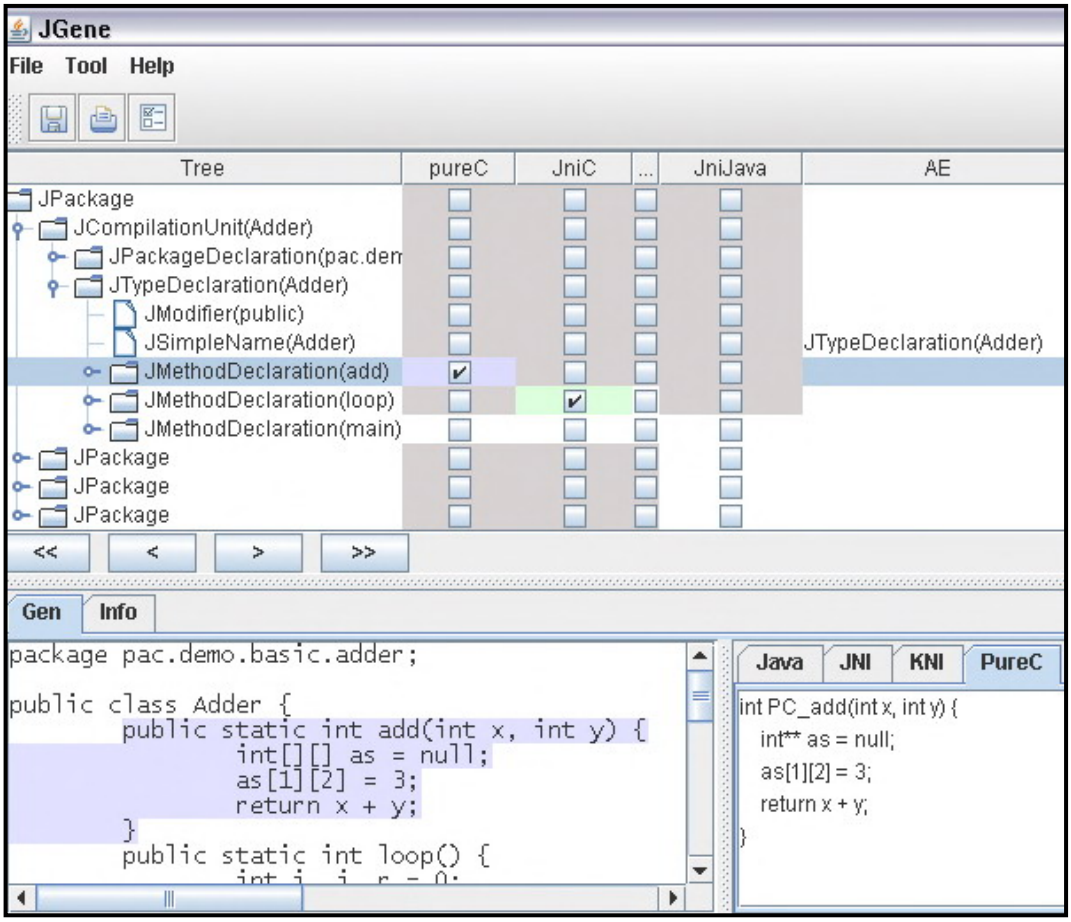


Figure 5. Sample using interface of JGene

Figure 5 above shows a snapshot of the JGene in use, using the example attributes described previously. Specifically, the user can examine the program structure and the attributes for each AST elements. Some choices are disabled due to various rules (written in Java). For the choices entered by the user, other choices may be filled or disabled accordingly. Thus although there seem many user-entered choices in Figure 5, the user in fact only decide, at method level, which methods are to be generated into pure C or JNI-compliant C.

For example, the ***PureC*** attribute may have associated rules defined by the developer:

- An element can be ***PureC***, ***JniC***, or ***Java***, exclusively.
- It cannot be ***PureC*** if ***UsesDefinedClassesOnly*** is false
- It is ***PureC*** when its parent is ***PureC***
- It is ***PureC*** if called by another ***PureC*** element
- It cannot be ***PureC*** if called by Java

All these rules are written as separate classes. In Figure 5, when the user chooses that the method add() is ***PureC***, the action triggers the re-evaluation of the method loop(), which should have value undecided at the time. According to the rule, loop() cannot be ***Java*** (hence the ***Java*** attribute becomes false), although it may still be ***PureC*** or ***PureC***, which is to be specified by the user later. Conversely, if the user select ***PureC*** for loop() first, then add() must be ***PureC*** according to the rules, thus its ***JniC*** and ***Java*** attributes all become false.

3.5. Code Generation

In this section we show some examples of program transformation using JGene. Before going into the example, we first describe briefly about the Java Native Interface (JNI). JNI is a programming framework that allows Java code running in Java VM to call and be called by native applications and libraries written in other languages such as C, C++ and assembly. To bind native method code to the Java application, a native method interface is used, which has been standardized across most JVMs. This interface is used to write native methods to handle situations when an application cannot be written entirely in the Java programming language such as when the standard Java class library does not support the platform-dependent features or program library. It is also used to modify an existing application, written in another programming language, to be accessible to Java applications.

For example, consider the pure java program below:

```
public class adder {
  void exec() {
    int x = add(1, 2);
    System.out.println(x);
  }

  int add(int i, int j) {
```

```
    return i + j;
  }
}
```

The corresponding JNI-compliant C program may be generated below:

```
// Java
public class adder {
  public void exec() {
    int x = add(1, 2);
    System.out.println(x);
  }

  public native int add(int i, int j);
}

// JNI
JNIEXPORT jint JNICALL Java_adder_add(JNIEnv *env, jclass cl, jint
  i, jint j) {
  return i+j;
}
```

In the context of embedded systems, the overhead to support JNI becomes large and may not justify the gain. The main reason is that JNI is intended to be standardized across multiple JVMs, so that the same native C/C++ code may still be portable, but the benefit of this approach is less relevant for embedded systems. KVM Native Interface (KNI) is a trimmed-down version of JNI for KVM - a small Java VM written for embedded systems. Unlike JNI, KNI is essentially implementation specific, not intend to be standardized. Another VM for embedded systems, CVM, is a larger JVM implementation that complies with the JNI standard, has more advanced JIT compiler technology inside, and hence requires more resources than KVM. Note that many of the standard library classes depend on KNI to provide functionality to the developer and the user, e.g. I/O file reading and sound capabilities. Including performance- and platform-sensitive API implementations in the standard library allows all Java applications to access this functionality in a safe and platform-independent manner.

Below we illustrate the transformation targeting KNI using the same example given previously. Currently, the transformation framework only works for a subset of input programs, namely, Java programs with certain syntactical restrictions. The goal is to investigate the potential usefulness of the transformation approach in general and the internals of the KVM and CVM. We show the generated KNI C program below.

```
// Java
public class Adder
{
```

```
  public void exec() {
    int x = add(1, 2);
    System.out.println(x);

  public native int add(int i, int j);
}


// KNI
KNIEXPORT KNI_RETURNTYPE_INT adder_add()
{
  jint i=KNI_GetParameterAsInt(1);
  jint j=KNI_GetParameterAsInt2);
  KNI_ReturnInt(i + j);
}

// Modified Java class
public class Test
{
  public native int loop(n);
}
```

As the example above shows, when targeting KNI, the generated C function needs to comply with the programming model defined by KNI. In particular, parameter passing is done through explicit stack operations using predefined KNI macros like KNI_GetParameterAsInt(). KNI also defines macros for accessing objects inside KVM. Note that this indicates some potential issues when the generated C function is to be placed on a different core than the one running the KVM.

Another more involved example is a Lens Blur Filter (see Chapter 4), as shown below:

```
public static int[] lensBlurFilter(int[] rgbIn, int width, int height)
{
 ...
 ImageFFT fft = new ImageFFT( Math.max(log2rows, log2cols) );

 int[] rgb = new int[w*h];
 ...
 // Create the kernel
 for ( int y = 0; y < h; y++ ) {
   for ( int x = 0; x < w; x++ ) {
     ...
   }
 }

 // Normalize the kernel
 i = 0;
 for ( int y = 0; y < h; y++ ) {
   for ( int x = 0; x < w; x++ ) {
     mask1[i] /= total;
     i++;
   }
 }

 fft.transform2D( mask1, mask2, w, h, true );

 for (...) {
   for (...) {
```

```
      ...
      //src image getRGB
      ...
    }
  }

  // Transform into frequency space
  fft.transform2D( ar1, ar2, cols, rows, true);
  fft.transform2D( gb1, gb2, cols, rows, true);

  // Multiply the transformed pixels by the transformed kernel
  ...
  // Transform back
  fft.transform2D( ar1, ar2, cols, rows, false );
  fft.transform2D( gb1, gb2, cols, rows, false );
  ...
  //dst Image setRGB
  ...

  Return ...;
}


public void transform2D( float[] real, float[] imag, int cols, int
  rows, boolean forward )
{
  ...
  // FFT the rows
  for ( int y = 0; y < rows; y++ ) {
    ...
    transform1D(rtemp, itemp, log2cols, cols, forward);
    ...
  }

  // FFT the columns
  for ( int x = 0; x < cols; x++ ) {
    ...
    transform1D(rtemp, itemp, log2rows, rows, forward);
    ...
  }
}
```

The most frequently invoked method in the Lens Blur Filter is *transform2D()*. Indeed, after profiling, we found that *transform2D()* occupied most of the run time. Naturally, we would like to make *transform2D()* native, although we could also translate all *lensBlurFilter()*, including *transform2D()* and the other methods it calls into pure C code. But this may result in excessive code size. Using JGene, we are able to explore both approaches quickly. Chapter 4 will show the performance results.

Consider the case where only *transform2D()* is made native. The following code shows that *transform2D()* is generated in KNI-compliant format and *transform1D()* in pure C format.

```
KNIEXPORT KNI_RETURNTYPE_VOID
  Java_ccl_midlet_image_ImageFFT_transform2D()
{
  /* Get the java input parameter */
```

```
  KNI_StartHandles(5);

  KNI_DeclareHandle(handle1);
  KNI_GetParameterAsObject(1, handle1);
  jint handle1_len = KNI_GetArrayLength(handle1);
  jfloat real [handle1_len];
  KNI_GetRawArrayRegion(handle1, 0, handle1_len*sizeof(jfloat),
  (jbyte*)real);
  ...
  jint cols = KNI_GetParameterAsInt(6);
  jint rows  = KNI_GetParameterAsInt(7);
  jboolean forward  = KNI_GetParameterAsBoolean(8);
  // start real java code here
  ...

  // FFT the rows
  for (int y = 0; y < rows; y++) {
    ...
    transform1D(rtemp, itemp, w1, w2, w3, log2cols, cols, forward);
    ...
  }

  // FFT the columns
  for (int x = 0; x < cols; x++) {
    ...
    transform1D(rtemp, itemp, w1, w2, w3, log2rows, rows, forward);
    ...
  }
  // end real java code
  KNI_SetRawArrayRegion(handle1, 0, handle1_len*sizeof(jfloat),
  (jbyte*)real);
  KNI_SetRawArrayRegion(handle2, 0, handle2_len*sizeof(jfloat),
  (jbyte*)imag);

  KNI_EndHandles();
}

void transform1D(float *real, float *imag, float *w1, float *w2, float
  *w3, int logN, int n, bool forward)
{
  scramble(n, real, imag);
  butterflies(n, logN, forward ? 1 : -1, real, imag, w1, w2, w3);
}
```

Note that static program transformation also has intimate relation with the dynamic JIT compiler. For example, in some situation, a method may become too big to compile dynamically by the JIT compiler, because the embedded system may not provide sufficient memory. One example is the FFT benchmark in Scimark2 (see Chapter 4). In this case we try to rewrite the main method of *transformInternal()*, the goal is try to reduce the original method code size by dividing *transformInternal()* into several smaller methods. With the help of JGene, the user can easily perform the necessary refactoring and explore the differences in terms of memory cost and speed-ups. In our experiment, after proper refactoring, the new version of *transformInternal()* could be translated into native format easily, and the original *transformInternal()* can be JIT-compiled again.

Although the JGene offer an automatic translation mechanism, it does not guarantee the translated code work exactly the same as the original code. The engineer should look at the generated code along with its Java counterpart to ensure that the translation scheme is valid. Because of this, no extensive compiler background needed to extend and use JGene unless one wants to provide more advanced analyses.

Chapter 4. Experiments

For general benchmarking, we use the SciMark2 benchmark, which is a composite Java benchmark measuring the performance of numerical codes occurring in scientific and engineering applications. It consists of five computational kernels which are chosen to provide an indication of how well the underlying JVM/JIT compilers perform on applications utilizing these types of algorithms. SciMark2 includes the following kernels:

- **Fast Fourier Transform (FFT):** performs a one-dimensional forward transform of 4K complex numbers. This kernel exercises complex arithmetic, shuffling, non-constant memory references and trigonometric functions.
- **Successive Over-relaxation (SOR):** on a 100x100 grid exercises typical access patterns in finite difference applications, for example, solving Laplace's equation in 2D with Drichlet boundary conditions.
- **Monte Carlo:** integration approximates the value of Pi by computing the integral of the quarter circle y = sqrt(1 - x^2) on [0,1]. It chooses random points with the unit square and compute the ratio of those within the circle.
- **Sparse Matrix Multiply:** uses an unstructured sparse matrix stored in compressed-row format with a prescribed sparse structure.
- **Dense LU matrix factorization:** Computes the LU factorization of a dense 100x100 matrix using partial pivoting.

As mentioned, our framework relies primarily on JNI and/or KNI to connect the virtual machine with external C modules.

4.1. Simple Benchmarking

For experiment purpose, we developed a CCL_PKG package alongside with other JSRs in the PhoneMe CLDC/MIDP distribution, where the native Java methods are also placed in this package. The framework is streamlined so that after the user makes transformation decisions, the resulting programs, including Java and C ones, will be placed in suitable places and the whole distribution can be rebuilt.

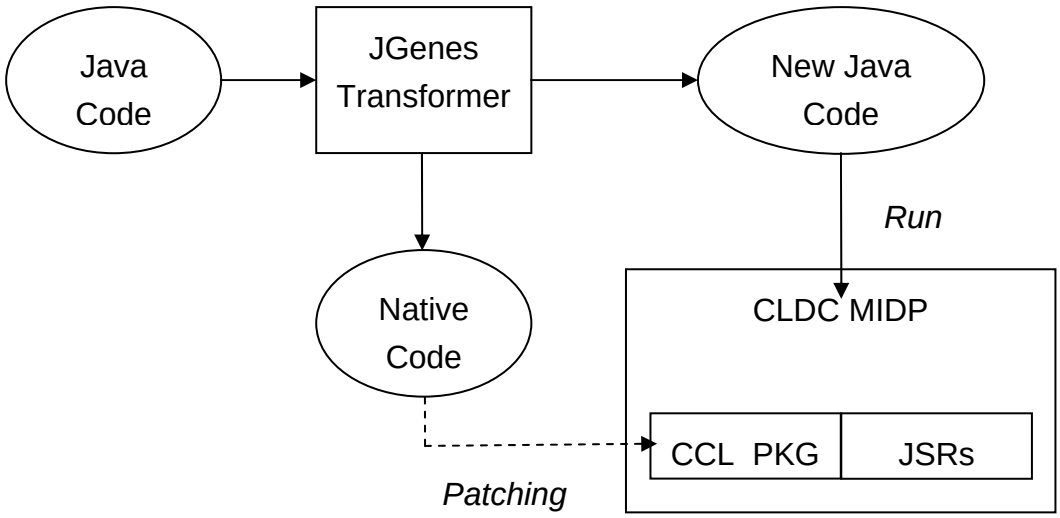


Figure 6. Relation between JGenes and CLDC/MIDP

Note that there is an adaptive, just-in-time (JIT) compiler in the CLDC HotSpot Implementation virtual machine. The CLDC JIT compiler is a one-pass compiler that provides a number of basic optimizations. The compiler makes a preliminary scan of a method to determine entry points, and then a target-dependent optimizer makes a final pass through the generated code.

Here we develop a sample test suite to compare run-time performance with or without AOT transformations. In this test suite we use simple routines such as prime search, recursion Fibonacci, recursion accumulator and 2D matrix multiplication. Detailed description of the test suite is given in Appendix. The result is shown in Table 1. (Time unit is **millisecond**, **PrimSearch** range is 1~50000, **Fibonacci** N=42, **Accumulator** N=2500 Cycle=100000, **Matrix** is 256x256 double array).

Table 1. The performance results on various configurations

Type	CLDC/JIT	CLDC/No JIT	CLDC/KNI	CVM/JNI	GCJ
PrimSearch	5999	17138	4037	4238	4237
Fibonacci	15192	96302	7315	9539	9491
Accumulator	5606	29128	185	913	1811
MatrixMulti	13635	14005	59	59	127
MatrixMulti (Native product)	700	827	--	404	--

Figure 7 shows the performance results for matrix multiplications. We found that the CLDC/KNI case gets better performance because the critical code becomes native. The CLDC/JIT case uses the JIT compiler to speedup the execution, but it has almost no improvement for matrix multiplications. After further investigation, we found that the problem is due to its slow execution for floating numbers. In the CLDC/KNI case, on the other hand, the C compiler helps to generate more efficient code for floating point operations. Thus we found the performance improvement for CLDC/KNI from 13.635 sec to 59 msec. On the other hand, if we isolated the dot product operation to a new method and running in native, we still get a great improvement for CLDC/JIT with this re-factored Java code. This also shows the advantage of our framework that helps developers in exploring various combinations of AOT and JIT compilations strategies.

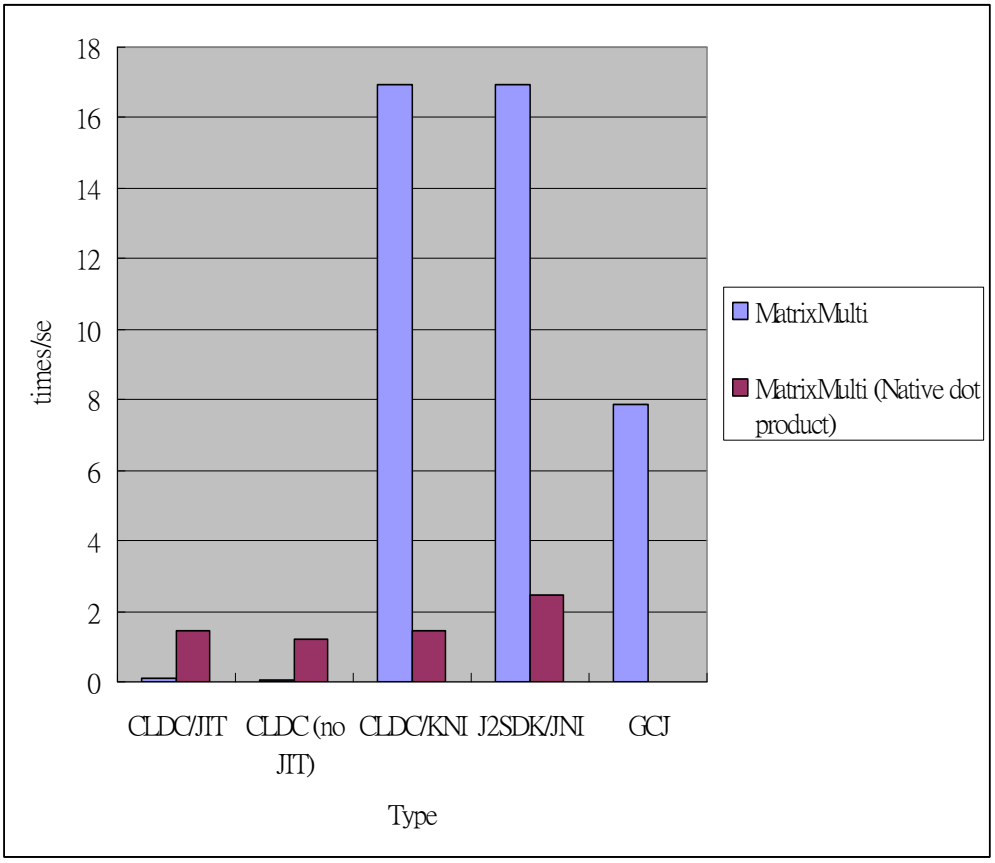


Figure 7. The 256x256 matrix multiplication execute result

Figure 8 shows the results for recursive calls (recursive accumulator and Fibonacci function). CLDC/JIT could not handle too many recursive calls due to limited heap and stack sizes, even it is better than without using JIT compiler. The big improvement demonstrated by the CLDC/KNI and CLDC/JNI cases is due to the fact that with –o3 flag turned on, the GCC

compiler performs tail recursion optimization when applicable. Even when recursive calls are not in a tail-recursion form, such as in the Fibonacci test case, CLDC/KNI still reduce the execution time more than the CLDC/JIT case (7.315 vs. 15.192 times per second) without increasing the code size.

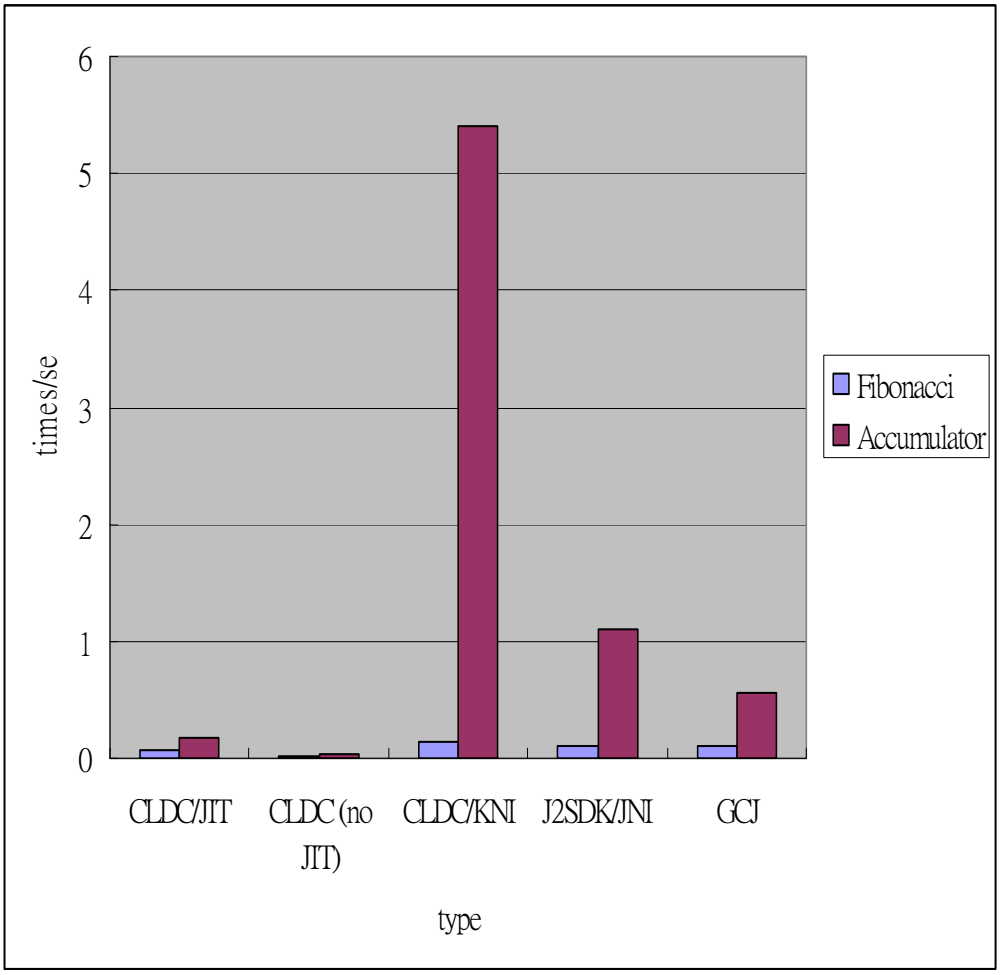


Figure 8. The speedup of GCC optimization in recursion

4.2. Scimark2 Benchmarking

As indicated before, Scimark2 is an open source benchmark for Java. In this experiment we translated the five functions in Scimark2 into native C programs. For example, a core method called measureFFT() is shown below, in which the method transformInternal() is made native:

```
native static void transformInternal(double data[], int direction);

public static double measureFFT(int N, double mintime, Random R)
{
  ...
  while(time slot) {
    ...
    for (int i=0; i<cycles; i++) {
      FFT.transform(x);  // forward transform
      FFT.inverse(x);    // backward transform
    }
    ...
  }
  ...

  // approx Mflops
  return FFT.num_flops(N)*cycles/ Q.read() * 1.0e-6;
}

public static void transform(double data[]) {
  transformInternal(data, -1);
}

public static void inverse(double data[]) {
  transformInternal(data, +1);
  // Normalize
  int nd=data.length;
  int n =nd/2;
  double norm=1/((double) n);
    for(int i=0; i<nd; i++)
      data[i] *= norm;
}

// KNI part
KNIEXPORT KNI_RETURNTYPE_VOID
  Java_scimark_org_FFT_transformInternal()
{
  //do real fft transformation
  ...
}
```

The performance results for various Scimark2 benchmarks are shown in Table 2 on our x86 testing platform:

Table 2. The result of Scimark2 scores on CLDC and CDC platforms

Type	CLDC/JIT	CLDC/KNI	CDC(no JIT)	CDC/JIT
FFT	1.115	47.982	3.402	9.850
SOR	2.260	404.822	10.846	20.420
Monte Carlo	1.388	26.715	2.417	3.010
Spares Matmult	2.206	517.886	5.951	21.141
LU	2.151	334.476	6.902	17.380

As shown in Figure 9, the improvement achieved via native method is significant; only the **Monte Carlo** case performs relatively weak compared to the other cases. The reason is due to excessive use of Java objects.

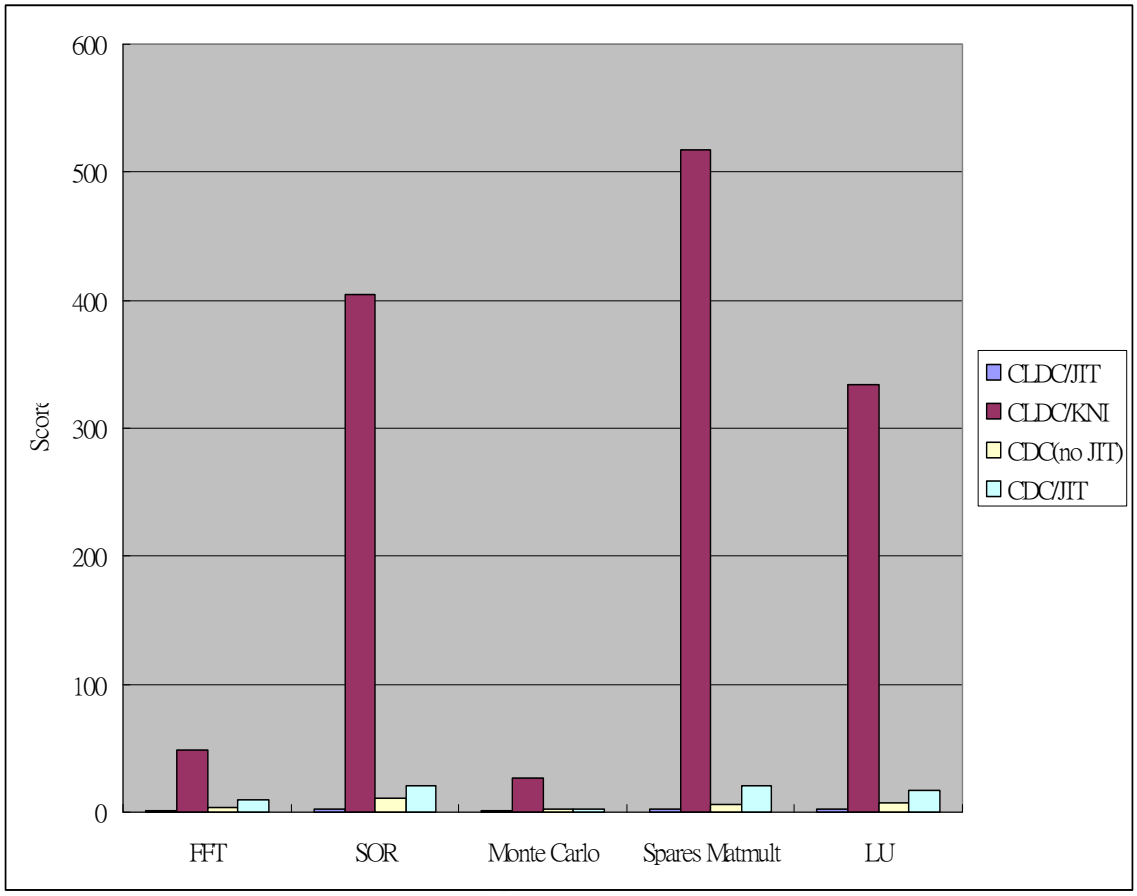


Figure 9. The result of CLDC/KNI Scimakr2 scores

4.3. Code Refactoring

In this experiment we use the FFT benchmark in Scimark2 to illustrate the benefit of refactoring combined with Java native support. As described previously, FFT invokes *transformInternal()* extensively. The original *transformInternal()* is sketched below:

```
protected static void transformInternal(double data[], int
 direction)
{
 ...
 bitreverse(data);

 /* apply fft loop */
 for (int bit = 0, dual = 1; bit < logn; bit++, dual *= 2) {
   ...
   /* a = 0 */
   for (int b = 0; b < n; b += 2 * dual) {
```

```
    ...
  }

  /* a = 1, (dual-1) */
  for (int a = 1; a < dual; a++) {
    for (int b = 0; b < n; b += 2 * dual) {
      ...
    }
  }
 }
}
```

After refactoring, the inner loop is changed to fftLoop(), as indicated below:

```
protected static void transformInternal (double[] data, int
 direction)
{
 ...
 bitreverse(data) ;

 /* apply fft loop */
 for (int bit = 0, dual = 1; bit < logn; bit++, dual *= 2) {
   fftLoop(data, n, dual, direction);
 }
}
```

This kind of refactoring is worse than the original one in general, provided that the JVM has abundant resources and powerful JIT compilation. For embedded systems, however, the original method maybe too large to compile, while in the refactored version the fftLoop() may still be dynamically compilable. When AOT transformation is concerned, it is also possible that the original method cannot be translated due to many rules (e.g. some methods it calls are external classes), yet in the refactored case the fftLoop() is simple enough to translate, as shown below:

```
KNIEXPORT KNI_RETURNTYPE_VOID Java_FFTRefactor_fftLoop() {
 ...
 /* a = 0 */
 for (int b = 0; b < n; b += 2 * dual) {
   fftLoopA0(data, b, dual);
 }

 /* a = 1 */
 for (int a = 1; a < dual; a++) {
   fftLoopA1(data, n, a, dual, direction, w_real, w_imag);
 }
 ...
}
```

The java modified FFT test performance results are given in Table 3 (FFT **size**=8, **cycles**=70000, **unit**=msec):

Table 3. The effort of refactoring FFT

Type	No JIT	CLDC/JIT	CLDC/JIT (Mini cache)	CLDC/KNI
Original FFT	9755	9205	16067	701
Refactor FFT	10709	9608	10083	871

The results indicated that with static program transformation still outperform JIT compilation. More interestingly, it is also shown that programs will smaller methods may work more stably across different configurations than those that contain large blocks of code. This is seen in the “Mini cache” case in which we limit the CLDC JIT code cache to be quite low such that the original big method cannot be compiled into CLDC JIT cache, and the attempt is tried many times.

4.4. Image Filter Benchmarking

To test our approach for more general applications, we developed a simple image filter toolkit which offers several simple filters. The main functions of this tool are gray, sobel, box blur, and lens blur filters. The first three filters are space domain array masks. All space domain filters work by sliding a rectangular array of numbers over target image. This array is called the convolution kernel. For every pixel in the image, we take the corresponding numbers from the kernel and the pixels they are over, multiply them together and add all the results together to make the new pixel. The lens blur filter is handled by performing frequency-domain Discrete Fourier Transform successively. In this experiment we focus on these kernel operations. The run time results are showed in Table 4 (The sample image size is 240x320 pixels).

Table 4. The image filter sample tests result

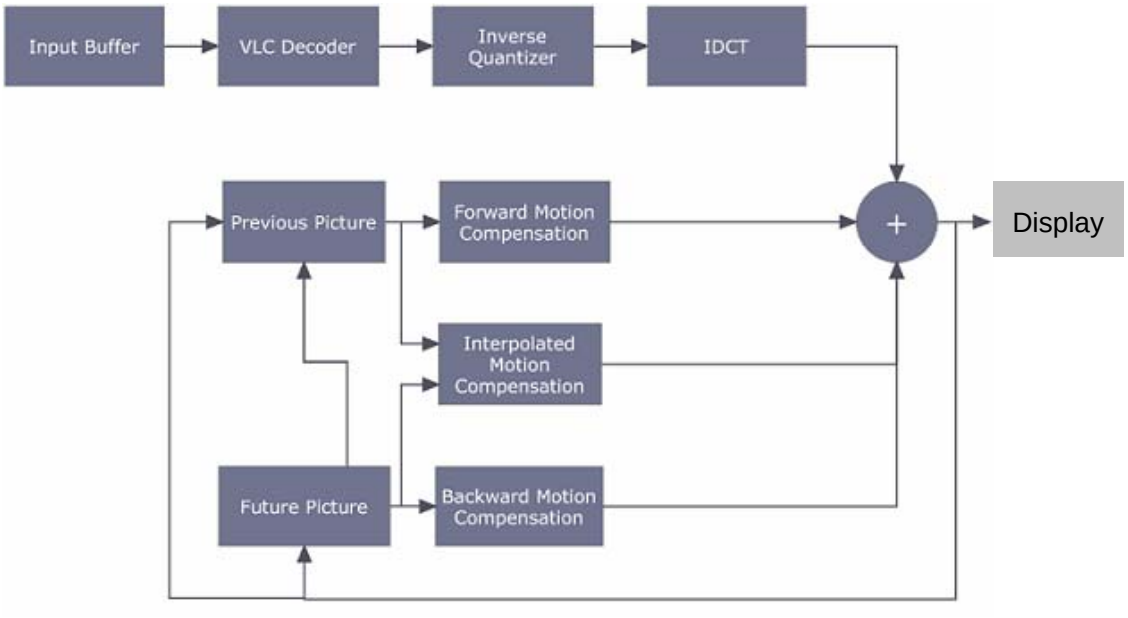
Type	CLDC/JIT	CLDC(no JIT)	CLDC/KNI
Gray(1K cycle)	3987	55671	643
Sobel(1K cycle)	5622	192139	2656
Box Blur(1K cycle)	3793	251430	2489
Len Blur	21941	22947	111
Len Blur FFT/KI	1197	1211	-

We found that CLDC/JIT gets a big improvement in first three filters but performs badly for Len Blur. The reason is the same as which discussed before, that the Len Blur filter does many floating point operations, revealing the poor handling of floating point instructions in CLDC/JIT. Since the Len Blur invoking FFT many times, we still could get much better performance than original if we only translated the FFT method. Figure 10 shows the results more clearly.

Figure 10. The result of our image filter benchmarking

4.5. MPEG Decoder Benchmarking

J2ME mpeg decoder is our mpeg decoder test application. It is an open source suite (http://wiki.forum.nokia.com). The decoding algorithm is illustrated in Figure 11:



[From http://wiki.forum.nokia.com/index.php/J2ME]

Figure 11. The decoding flow of J2ME mpeg decoder

Specifically, the Bitmap Java class performs the conversion from Y'CbCr 4:2:0 to RGB. We use JGene to translate the transform() method into KNI method, improving the frame rate from 8 fps to 11 fps (sample file is 160x120).

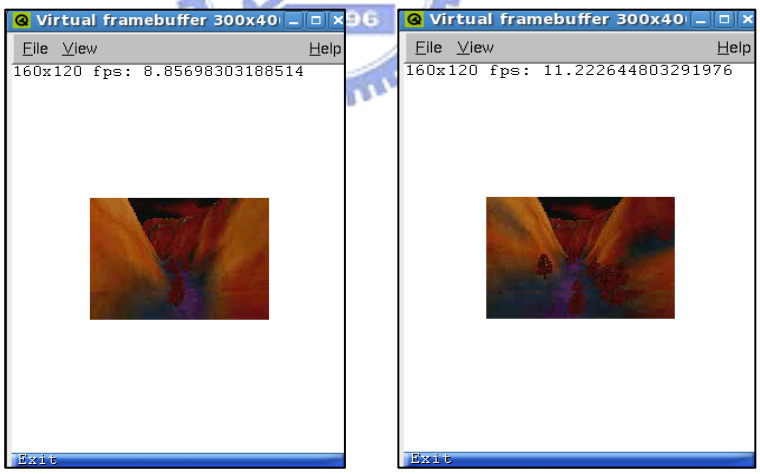


Figure 12. The mpeg display fps before(L) and after(R) *transform()*

4.6. Experiments on PAC and Inter-processor Communication

We have conducted several preliminary investigation and experiments on the PAC development board. As part of the PAC project goal is to construct an embedded Java

execution environment, we have integrated various components into a more stream-lined development environment. Figure 13 below illustrates the layout of the environment, where the toolchain is based on Scratchbox – a cross compilation toolkit. It also provides a full set of tools to integrate and cross compile an entire Linux distribution.

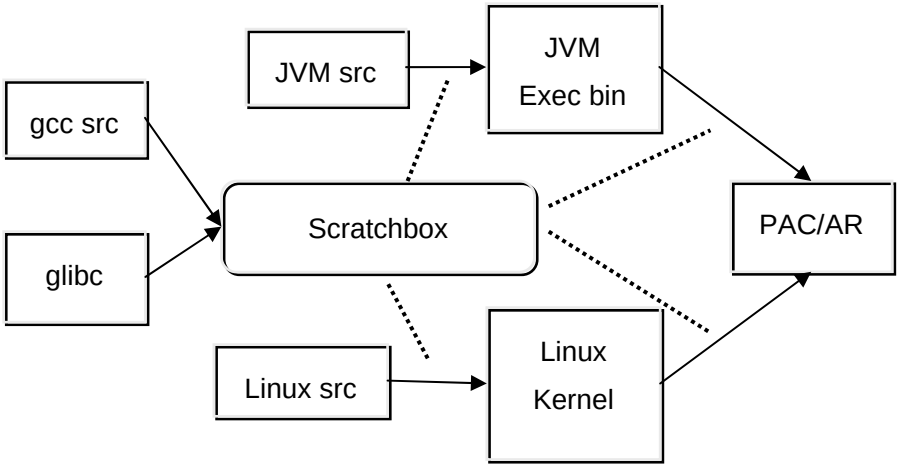


Figure 13. The build environment

As shown in the figure, platform-specific gcc and glibc ports (e.g. ARM-based *arm-none-linux-gnueabi-gcc*) are built with Scratchbox, which in turn becomes the build environment for Linux and JVM. In addition to ARM-based architectures, such as the emulator shipped with Scratchbox or PAC, other platforms such as Intel Xscale (IXDP425) combined with other operating systems such as (MontaVista) have also been used. Figure 14 and Figure 15 demonstrates some examples running on the PAC platform:

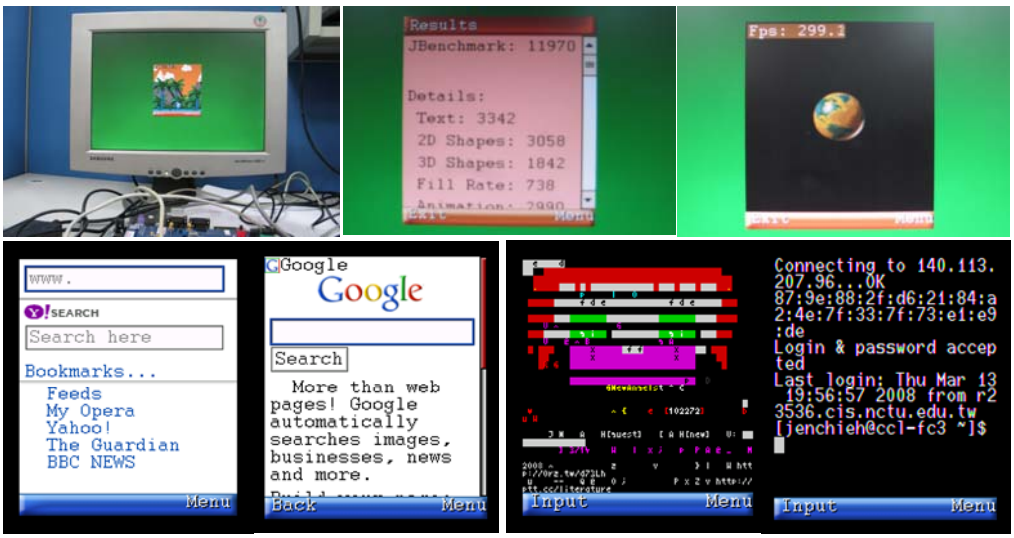
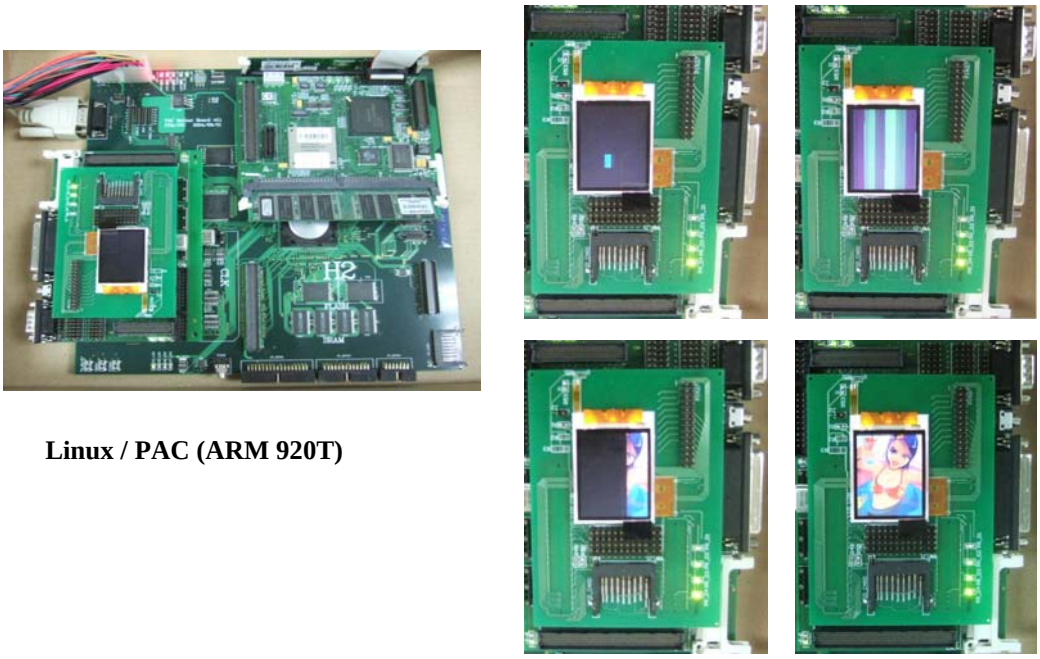


Figure 14. MIDP examples on PAC



Linux / PAC (ARM 920T)

Figure 15. Application examples on PAC LCD

On the PAC platform we have also perform many benchmarking experiments, including Scimark2, JBenchmark1, and JBenchmark2.

The PAC platform includes an implementation of inter-process communication (IPC) for bridging the MPU core and the DSP. Specifically, a micro kernel has been developed that provides simple task scheduling as well as interrupt-based programming interface. Such an IPC mechanism has many implications for the JGene transformer. Although the communication is interrupt-based, the programming model for IPC can be either synchronous or asynchronous. For the synchronous case, the micro kernel implementation includes an associated library allowing the programmer to communicate with the DSP core in an RPC style. For the asynchronous case, the programmer is responsible of creating suitable interrupt routines, and the main program (in the MPU side) needs to adopt an event-driven programming model. It is worth noting that with suitable generation rules, JGene can be used to cope with both IPC styles.

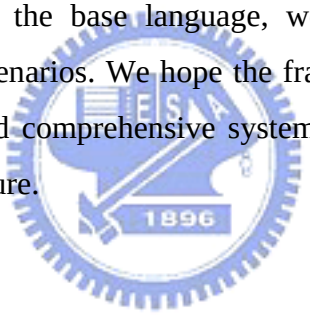
We have also developed some simple programs to test the IPC with the DSP core via micro kernel. Below are some examples:

```
// Modified Java class
public class Test {
   public native int loop();
}

// Generated C function on the ARM side
KNIEXPORT KNI_RETURNTYPE_INT Java_loop() {
    TASKID tid = pac_create(_PAC_loop, 2);
    pac_rpc(tid);
    pac_wait(tid);
    pac_read32(0x9000);
    KNI_ReturnInt(result);
}

// Generated C function on the DSP side
int PAC_loop() {
   int a, b, i, result = 0;
   for (i = 0; i < 10; ++i) {
     a = 3 + i; b = 4 - 2 * i;
     ...
   }

  //memory write with result for arm get
   *(unsigned int *) (0xC2009000) = result;

syscall(…); // trigger ARM
}
```

In the example above, for KVM to interact with the DSP core without modifying KVM itself, we generate a dispatcher function using KNI, and generate a C function managed by the micro kernel on the DSP core. Clearly, procedures about IPC can be placed in the dispatcher code, while the actual code for computation can be build by DSP-targeted compiler and run on the DSP core. It is possible to adjust the IPC parameters or even change to a different IPC model by altering the generation rule for the dispatcher code.

Chapter 5. Conclusion

In this thesis we argued the benefits of using program transformation to reduce the burden of developing applications for embedded systems. We have investigated research areas on program transformation and compiler techniques, with a focus on embedded software development. We showed that the design space to be explored is extremely large for a generator to generate suitable programs, and existing techniques often tackle only the design space partially. We proposed a transformation framework aiming at equipping engineers with flexible and extensible means to explore the design space more efficiently. We focus on transforming Java programs into a mixture of Java and platform-specific C programs, and develop associated tools to streamline the process. A near-term goal is to add profiling capability into JGene so that profiling information gathered from actual program execution can be used by JGene again to aid the user in exploring the design space. Despite the requirement of using Java as the base language, we believe our framework is already powerful for many practical scenarios. We hope the framework's extensible design can lend itself into a more powerful and comprehensive system that matches the need of embedded systems development in the future.



References

[Aarts] B. Aarts, M. Barreteau, F. Bodin, P. Brinkhaus, Z. Chamski, H.-P. Charles, C. Eisenbeis, J. Gurd, J. Hoogerbrugge, P.Hu,W. Jalby, P.Knijnenburg, M. O'Boyle, E. Rohou, R. Sakellariou, H. Schepers, A. Seznec, E. A. Sthr,M. Verhoeven, and H.Wijshoff, "OCEANS: Optimizing compilers for embedded HPC applications," Lecture Notes in Computer Science, August 1997.

[Ali] Ayaz Ali and Lennart Johnsson, Dragan Mirkovic, "Empirical Auto-tuning Code Generator for FFT and Trigonometric Transforms", Workshop on Optimizations for DSP and Embedded Systems, March 11, 2007.

[Alur] Rajeev Alur, Franjo Ivancic, Jesung Kim, Insup Lee, Oleg Sokolsky, "Generating Embedded Software from Hierarchical Hybrid Models", ACM SIGPLAN Notices, Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems LCTES '03, Volume 38 Issue 7. pp. 171-182

[Barat] F. Barat, T. Vander Aa, M. Jayapala, G. Deconinck, R. Lauwereins, and H. Corporaal, "Methodology for building processor design space exploration frameworks", 3th Workshop on Optimizations for DSP and Embedded Systems, March 20, 2005.

[Bennett] Richard Vincent Bennett Alastair Colin Murray Bjorn Franke Nigel Topham, "Combining Source-to-Source Transformations and Processor Instruction Set Extensions for the Automated Design-Space Exploration of Embedded Systems", LCTES'07 June 13–15, 2007. pp. 83-93

[Bravenboer] Martin Bravenboer, Karl Trygve Kalleberg, and Rob Vermaas, "Stratego/XT 0.16: Components for Transformation Systems", ACM/SIGPLAN Workshop Partial Evaluation and Semantics-Based Program Manipulation Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, 2006. pp. 95-99.

[Burgaard] Kim Burgaard, Flemming Gram Christensen, Jørgen Lindskov Knudsen, Ulrik Pagh Schultz, "Compiling Java for Low-end Embedded Systems", ACM SIGPLAN Notices , Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems LCTES '03, Volume 38, Issue 7. pp. 42-50

[Cavazos] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael F. P. O'Boyle, and Olivier Temam, "Rapidly Selecting Good Compiler Optimizations Using Performance Counters", International Symposium on Code Generation and Optimization (CGO), 2007. pp. 185-197

[Chen] C. Chen, J. Chame, and M. Hall, "Combining Models and Guided Empirical Search to Optimize for Multiple Levels of the Memory Hierarchy", 2005 International Symposium on Code Generation and Optimization (CGO), 2005. pp. 111-122

[Cheng] Allen Cheng, Gary Tyson, and Trevor Mudge, "FITS: Increasing Code Density for Embedded Systems with a Cost-Effective 16-bit Synthesis Technique", Workshop on Optimizations for DSP and Embedded Systems, March 21, 2004. pp. 920-923

[Cordy] James R. Cordy, "Source Transformation, Analysis and Generation in TXL", ACM/SIGPLAN Workshop Partial Evaluation and Semantics-Based Program Manipulation Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, 2006. pp. 1-11

[Dupre] Michael Dupre, Nathalie Drach, Olivier Temam, “VHC: Quickly Building an Optimizer for Complex Embedded Architectures”, International Symposium on Code Generation and Optimization (CGO), 2004. pp. 53-64

[Ernst] Ernst, R., “Codesign of embedded systems: status and trends”, *Design & Test of Computers*, Apr-Jun 1998, Vol. 15, No. 2. pp. 45-54

[Ertl] M.A. Ertl, “Stack caching for interpreters,” Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp.315–327, 1995.

[Franke] Björn Franke, Michael F. P. O'Boyle, John Thomson, Grigori Fursin, “Probabilistic source-level optimisation of embedded programs”, Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'05). pp. 78-86

[Fulton] Mike Fulton and Mark Stoodley, “Compilation Techniques for Real-Time Java Programs”, International Symposium on Code Generation and Optimization (CGO), 2007. pp. 221-231

[Grewal] Gary Grewal, S. Coros, “A Novel Scatter Search Procedure for Effective Memory Assignment for Dual-Bank DSPs”, Workshop on Optimizations for DSP and Embedded Systems, March 11, 2007.

[Hong] SungHyun Hong, *et al*, “Java Client Ahead-of-Time Compiler for Embedded Systems”, ACM SIGPLAN Notices , Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools LCTES '07, Volume 42, Issue 7. pp. 63-72

[JNI] Sun Microsystems Inc. *Java Native Interface Specification.* http://java.sun.com/products/jdk/1.2/docs/guide/jni/index.html

[Kim] Dngkeun Kim and Donald Yeung, “A Study of Source-Level Compiler Algorithms for Automatic Construction of Pre-Execution Code”, ACM Transactions on Computer Systems, Vol. 22, No. 3, August 2004, Pages 326–379

[Lambrechts] Andy Lambrechts, Praveen Raghavan, David Novo and Estela Rey Ramos, “Enabling WordWidth Aware Energy and Performance Optimizations for Embedded Processors”, 5th Workshop on Optimizations for DSP and Embedded Systems, March 11, 2007.

[Lee] Sheayun Lee, Jaejin Lee, Chang Yun Par, and Sang Lyul Min, “Selective Code Transformation for Dual Instruction Set Processors”, ACM Transactions on Embedded Computing Systems, Vol. 6, No. 2, Article 10, May 2007.

[Nacul] André C. Nácul and Tony Givargis, “Synthesis of time-constrained multitasking embedded software”, ACM Transactions on Design Automation of Electronic Systems, Vol. 11, No. 4, October 2006, pp. 822–847

[Ozturk] O. Ozturk, M. Kandemir, and S. W. Son, “ILP-based management of multi-level memory hierarchies”, Workshop on Optimizations for DSP and Embedded Systems, March 26, 2006.

[Pan] Zhelong Pan, Rudolf Eigenmann, “Fast and Effective Orchestration of Compiler Optimizations for Automatic Performance Tuning”, International Symposium on Code Generation and Optimization (CGO), 2006. pp. 319-332

[Panda] Panda *et al*, “Data and Memory Optimization Techniques for Embedded Systems”, ACM Transactions on Design Automation of Electronic Systems, Vol. 6, No. 2, April 2001, pp. 149–206

[Peri] Ramesh V Peri, Zino Benaissa, and Sri Doddapaneni, “A Binary Rewriting Tool for DSPs to Optimize Programs Based on Profile Information”, Workshop on Optimizations for DSP and Embedded Systems, March 21, 2004.

[Perkins] Michael Perkins, Graeme Roy, and Andrew Higham, “Profile-guided Optimization in the Cross-core Production C/C++ DSP Compilers”, Workshop on Optimizations for DSP and Embedded Systems, March 23, 2003.

[Sanghai05] K. Sanghai, D. Kaeli, and R. Gentile, Code and data partitioning on the Blackfin 561 dual-core platform, Workshop on Optimizations for DSP and Embedded Systems, March 20, 2005.

[Sanghai07] Kaushal Sanghai and David Kaeli, Alex Raikman and Ken Butler, “A Code Layout Framework for Embedded Processors with Configurable Memory Hierarchy”, Workshop on Optimizations for DSP and Embedded Systems, March 11, 2007.

[Shudo] Kazuyuki Shudo, Satoshi Sekiguchi, Yoichi Muraoka, “Cost-effective compilation techniques for Java Just-in-Time compilers”, Systems and Computers in Japan, Volume 35, Issue 12, pp. 10-24

[Suresh] Dinesh C. Suresh, Walid A. Najjar, Frank Vahid, “Profiling Tools for Hardware/Software Partitioning of Embedded Applications”, ACM SIGPLAN Notices, Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems LCTES '03, Volume 38 Issue 7. pp. 189-198

[Tabatabai] Ali-Reza Ad-Tabataba, Michal Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, James M. Stichnoth, “Fast, Effective Code Generation in a Just-In-Time Java Compiler”, SIGPLAN '98, pp. 280-290.[Vachharajani] Manish Vachharajani, Neil Vachharajani, and David August, Compiler Optimization-Space Exploration, Spyridon Triantafyllis, International Symposium on Code Generation and Optimization (CGO), 2003. pp. 204-215.

[VanderAa03] Tom Vander Aa, Murali Jayapala, Francisco Barat, Geert Deconinck, Henk Corporaal, and Francky Catthoor, “Software Transformations to Reduce Instruction Memory Power Consumption using a Loop Buffer”, Workshop on Optimizations for DSP and Embedded Systems, March 23, 2003.

[VanderAa05] T. Vander Aa, M. Jayapala, F. Barat, H. Corporaal, F. Catthoor, and G. Deconinck, “A high-level memory energy estimator based on reuse distance”, Workshop on Optimizations for DSP and Embedded Systems, March 20, 2005.

[Zhao03] Min Zhao, Bruce Childers, Mary Lou Soffa, “Predicting the Impact of Optimizations for Embedded Systems”, ACM SIGPLAN Notices , Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems LCTES '03, Volume 38 Issue 7. pp. 1-11

[Zhao05] M. Zhao, B. Childers, and M. L. Soffa, “A Model-based Framework: an Approach for Profit-driven Optimization”, 2005 International Symposium on Code Generation and Optimization (CGO), 2005. pp. 317-327

Appendix: Benchmarks used in Section 4.1

Prime Search:

```
public int primeSerach(long max) {
  int no = 0;
  for (long i = 0; i < max; i++) {
     if (isPrime(i)) {
         no = no + 1;
     }
  }
  return no;
}

private boolean isPrime(long i) {
  for (long test = 2; test < i; test++) {
     if (i % test == 0) {
         return false;
     }
  }
  return true;
}
```

Fibonacci Test:

```
public long fib(long i) {
  long res = calcFib(i);

  return res;
}

public long calcFib(long n) {
  if (n <= 1)
     return n;
  else
     return calcFib(n - 1) + calcFib(n - 2);
}
```

Accumulator Test:

```
public long accumulator(long n) {
  if (n == 0)
     return 1;

  return acc(n, 1);
}

private long acc(long n, long sum) {
  if (n == 1)
     return sum;
  else
     return acc(n-1, sum + n);
}
```

Matrix Multiplication test:

```
public void matrixMulti(int N) {
  Random R = new Random();

  double[][] A = new double[N][N];
  double[][] B = new double[N][N];
  double[][] C;

  for (int i = 0; i < N; i++)
     for (int j = 0; j < N; j++)
         A[i][j] = R.nextDouble();

  for (int i = 0; i < N; i++)
     for (int j = 0; j < N; j++)
         B[i][j] = R.nextDouble();
  C = new double[N][N];
  // Order: jik optimized ala JAMA


  double[] bj1 = new double[N];
  for (int j = 0; j < N; j++) {
     for (int k = 0; k < N; k++)
         bj1[k] = B[k][j];
     for (int i = 0; i < N; i++) {
         C[i][j] = dotProduct(A[i], bj1, N);
     }
  }
}

double dotProduct(double[] ai, double[] bj, int N) {
  double s = 0;
  for (int k = 0; k < N; k++) {
     s += ai[k] * bj[k];
  }

  return s;
}
```