Towards a Very High Bandwidth Wireless Battery Powered Device

John Glossner, David Routenberg,
Erdem Hokenek, and Mayan Moudgill
Sandbridge Technologies
Brewster, NY 10509

Michael J. Schulte and Pablo I. Balzola
EECS Department
Lehigh University
Bethlehem, PA 18015

Stamatis Vassiliadis
Computer Engineering Laboratory
Delft University of Technology
Delft, The Netherlands

Abstract

We discuss the hardware and software challenges in building a 2 Mbit per second wireless battery powered communications device. Of primary importance is power dissipation. To achieve aggressive power targets, a host of new techniques are required at all levels of the design hierarchy. Techniques for parallelizing saturating arithmetic will become important because of the software optimizations they enable. Highly configurable programmable structures will enable multiprotocol SOC solutions. To program complex SOCs, new compiler techniques will be required. Hardware implementations will need to be intimately aware of these software techniques. In particular, both signal processing code written in C and control code written in Java will drive new compilation techniques to enable broadband 3G wireless systems.

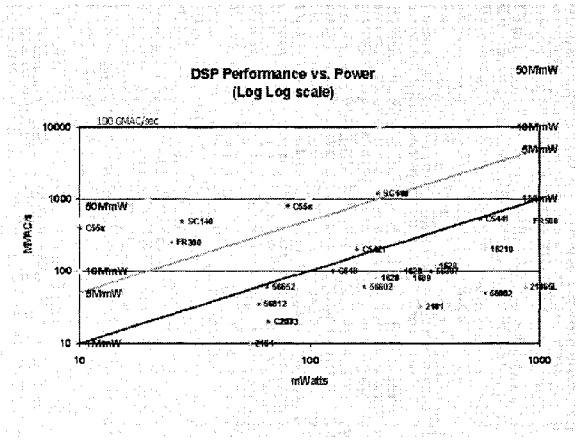

Figure 1. DSP Performance vs. Power

1. Introduction

High-speed communications are proliferating[10] and digital signal processors (DSPs) are accelerating this trend. DSPs have become a ubiquitous enabler for integration of audio, video, and communications[9]. Furthermore, DSPs are the driving force accelerating wireless communications.

Figure 1 shows DSP performance in Millions of Multiply Accumulates per Second (MMACs) vs. Power in milliWatts (mW). It is evident that most DSPs fall below the 1 MMAC/mW line. A few recent DSP announcements have pushed the power/performance limits to nearly 50 MMAC/mW.

Tremendous hardware and software challenges exist to realize a 2Mbps wireless battery powered device. First, power dissipation constraints are requiring new techniques at every stage of design - architecture, microarchitecture, software, algorithm design, logic design, circuit design, and process design. With performance requirements exploding as bandwidth demand increases, power conscious design becomes more difficult. SOC integration and low voltage process technologies will contribute to lower power system-on-a-chip (SOC) integrated circuits (ICs) but are insufficient as the only solution for streaming multimedia.

Second, DSP applications are becoming more complex. In wireless communications, GSM and IS-54 data rates were limited to less than 15 Kbps. Future third-generation (3G) systems may provide data rates more than 100 times the previous rates. Higher communication rates are accelerating higher DSP processing requirements. Complexity is driving the need to program applications in high-level languages. In the past, when only small kernels were required

0-7695-1056-6/01 $10.00 © 2001 IEEE

to execute on a DSP, it was acceptable to program in assembly language. Today, resource constraints prohibit these practices.

Third, Java may become the dominant programming paradigm for 3G systems. NTT DoCoMo recently rolled out Java-based services for its cellular subscribers and hardware solutions for efficient Java execution are being proposed[37].

Fourth, unlike many past developments, hardware designers will need to understand the complexities of software systems so that compilation techniques can be effective. With a large number of standards both existing and proposed for wireless communications, a programmable platform will be required for timely implementation.

Fifth, embedded and DSP wireless applications have distinct requirements when compared with general purpose processors [22]. The predominant algorithmic difference is that inner loops are easily described as vectors of moderate length. A key point is that the native datatype is often fixed-point fraction. This is in distinct contrast to general purpose processors (and most high-level languages) which operate on integer datatypes.

Finally, in addition to algorithmic differences, most DSPs are deployed in embedded environments where real-time constraints are prevalent. Real-time behavior has a dominant influence in the design of DSPs [28]. Whereas general-purpose applications can often manage with variable latency response, DSP applications, in contrast, should be able to precisely guarantee the latencies within the system.

In this paper we explore both hardware and software challenges in designing a high performance ultra-low power 2 Mbps wireless battery operated device. In the first section we explore hardware techniques for accelerating wireless systems. In the second section we explore software techniques for developing very complex applications. We then conclude with some directions we anticipate to be important for ubiquitous communications.

2. Hardware Productivity

Hardware advances for power efficient computing are critical to realizing very high bandwidth battery operated devices. The constraints on power consumption in these systems place a heavy burden on hardware designers. Compounding this difficulty is the fact the applications are becoming more complex and require sophisticated software productivity. In this section we describe two hardware techniques that enable higher software productivity.

2.1 Parallel Saturating Arithmetic Units

Many DSP applications, including GSM speech coders, perform millions of saturating arithmetic operations per second [20]. With saturating arithmetic, a result that is too large or too small to represent is saturated to the most positive or most negative representable number, respectively [22]. GSM speech coders and several other DSP applications have inner loops that perform saturating dot products on long vectors, with saturation after each arithmetic operation.

The GSM standards require that the results produced by GSM speech coders be identical to the results produced when the operations are performed serially [8]. Since saturating arithmetic operations are not associative, high-performance DSPs with several MAC units have to execute these operations sequentially to maintain GSM compliance. This significantly degrades performance and energy efficiency, since additional cycles are needed to guarantee GSM compliant results [36].

Hardware support for parallel saturating arithmetic improves the performance, energy efficiency, and numerical integrity of many DSP applications, by giving DSP compilers the opportunity to parallelize loops that contain saturating arithmetic operations, while maintaining bit-exact results. In [36], designs are presented for a saturating adder, multiplier, single MAC unit, and dual MAC unit. Since each of these units has only one fast carry-propagate adder on the critical path, they can execute in a single cycle.

The dual MAC unit presented in [36] performs two saturating MAC operations in parallel and accumulates their results with saturation. Saturation detection and selection logic ensures that the output of the dual MAC unit is identical to the result of the operations performed serially with saturation after each multiplication and each addition. Although this approach works well for two parallel MAC units, increasing the number of parallel MAC units beyond two significantly increases area, delay, and power dissipation.

In [29], an alternative approach is presented for designing processors that perform saturating dot products. In the first cycle, m saturating multipliers compute saturated products in parallel. In the next cycle, an $(m+1)$-input saturating multioperand adder (SMA) combines the outputs from each of the multiply units, and m new saturated products are computed. By pipelining the design and adding a feedback path, m additional elements of a saturating dot product can be generated and added every cycle. This approach is shown in Figure 2. Each saturating multipliers computes

$$P_{i+1} =< X_i \cdot Y_i > \qquad (\text{for } 1 \leq i \leq m) \qquad (1)$$

where $< R >$ indicates that R is saturated. The SMA com-

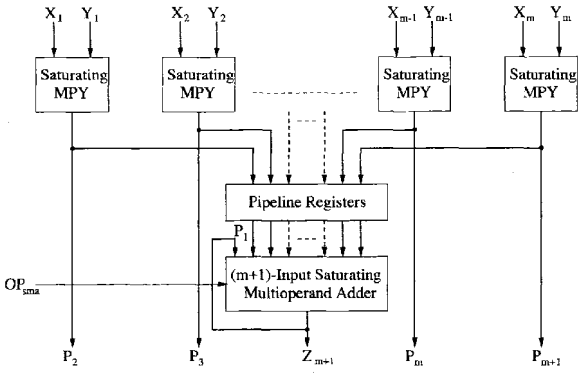

Figure 2. Parallel Saturating Arithmetic Units.

putes

$$Z_{m+1} = << \ldots < P_1 + P_2 > + \ldots + P_m > + P_{m+1} > \quad (2)$$

When the first m saturated products are added, P_1 is set to zero. In the remaining cycles, P_1 is set to Z_{m+1} from the previous computation. Thus, a p-element saturating dot product completes in $1 + \lceil p/m \rceil$ cycles.

Figure 3 shows the design of an optimized n-bit, 5-input SMA that uses the design methodology presented in [29]. The SMA is designed so that it has only one fast carry-propagate adder (CPA) on its worst case delay path, yet produces the same results that would be obtained if saturation were performed after each addition. An optimized m-input SMA units, uses $(m-1)$ V-GEN units, $(m^2 - 3m + 2)/2$ carry-save adders (CSAs), and $(m-1)$ CPAs to compute the m temporary sums, T_1 to T_m, as

$$T_i = V_i + \sum_{j=i+1}^{m} P_i \quad (3)$$

where $V_i = sp_i\ sp_i\ sp_i \cdots sp_i$, sp_i is the sign-bit of P_i, and $V_1 = P_1$. It also uses $(m^2 - 3m + 2)/2$ sign-detection circuits (SDCs), $(m-1)$ overflow detection logics (ODLs), $(m^2 - 3m + 2)/2$ 1-bit Muxes, and $(m-1)$ n-bit Muxes to determine which saturated additions overflow and to select the correct temporary sum. The signal o_i indicates that overflow occurs after the i^{th} saturated addition. The worst case delay path is equal to the delay of $(m-2)$ CSAs, plus one CPA, plus $(m-1)$ Muxes, plus one ODL.

To decrease the worst case delay of the SMA, the feedback value $P_1 = Z_{m+1}$ can be kept in carry-save format until the entire dot product is computed. This approach is shown in Figure 4, where the feedback value Z_5 is represented using a sum vector ZS_5, a carry vector ZC_5, and a sign-bit sz_5. With this approach, the CPAs are changed to SDCs, the feedback register and result selection Muxes

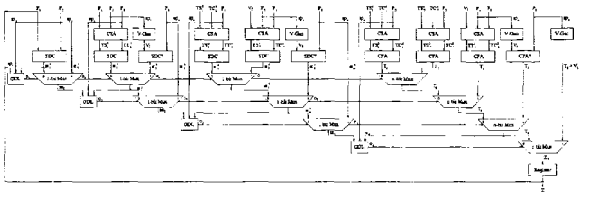

Figure 3. Optimized 5-Input Saturating Multi-operand Adder.

grow from n bits to $2n+1$ bits, and an additional CSA and CPA are required. Since the SDC plus the additional CSA have less delay than a CPA, the overall delay is reduced.

Area and delay estimates were made for 32-bit versions of the optimized 5-input SMA and the optimized 5-input SMA with carry-save feedback (SMA-WCSF) using VHDL models, LSI Logic's 0.6 micron LCA300K gate array library, and the Leonardo synthesis tool from Exemplar Logic. The optimized 5-input SMA requires 11,097 equivalent gates and has a worst case delay of 12.76 ns. The optimized 5-input SMA-WCSF requires 12,578 equivalent gates and has a worst case delay of 10.92 ns. A 16-bit saturating MAC unit implemented in the same technology requires equivalent 10,873 gates and has a worst case delay of 12.08 ns. Compared to the 5-input SMA, the 5-input SMA-WCSF requires 13.3% more equivalent gates and has 14.4% less delay.

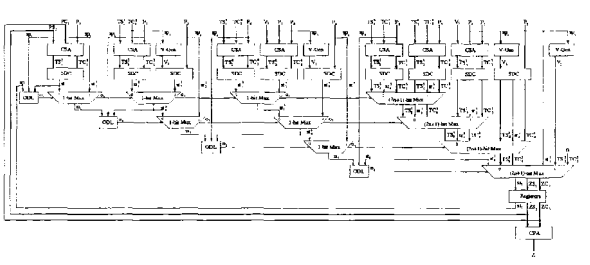

Figure 4. Optimized 5-Input Saturating Multi-operand Adder With Carry-Save Feedback.

2.2 Reconfigurable Computing

Diverse protocol requirements are driving the need for flexibility in hardware. Such versatility is a necessary property to accelerate communication and data transport protocols. Future wireless systems will be required to execute previous standards as well as multiple 3G standards. This requires a programmable approach.

Various approaches to programmability include automatic generation of processors and toolsets, hardware units that

can be reconfigured for a family of functions, and dynamic FPGAs which can morph their functionality based on the task being executed. Because of their flexibility, reconfigurable processors will be utilized where dynamic range varies significantly and in applications where the precision of operations is non-standard. In the longer term, if compiler technology can be exploited, FPGA fabrics such as PipeRench have an opportunity to significantly influence present computing paradigms[15].

3 Software Productivity

In the past, when only small kernels were required to execute on a DSP, it was acceptable to program in assembly language. Fifteen years ago, DSP applications only required about one thousand lines of C code. However, future generations of DSP applications, particularly 3G wireless, are anticipated to require one hundred thousand lines of C code. The rate of complexity is increasing 10x every 10 years. If a particular DSP is 70% compilable, that implies 30,000 lines of C code must still be hand translated into assembly code. Even at 90% compilability, the number of lines of C code to be hand translated is still equivalent to the complexity of DSP applications of just a few years ago. In the future, when the first 1 million line DSP application appears, it will no longer be feasible to consider any hand programmed assembly language DSP.

3.1 DSP Compilation Issues

There are a number of issues that must be addressed in designing a DSP compiler. First, there is a fundamental mismatch between DSP datatypes and C language constructs. A basic data-type in DSPs is a saturating fractional fixed-point representation. C language constructs, however, define integer modulo arithmetic. This forces the programmer to explicitly program saturation operations. The compiler must then deconstruct these idioms to recognize the underlying fixed-point operations.

A second problem for compilers is that previous DSP architectures were not designed with compilability as a goal. Future hardware techniques will require intimate coupling with software techniques to provide for efficient DSP compilation.

3.2 DSP Compilation Techniques

A number of solutions have been proposed to ameliorate the DSP compilation problem. First, special language extensions have been proposed [21, 23]. Typical additions may include special type support for 16-bit data types (Q15 formats), saturation types, multiple memory spaces, and SIMD parallel execution support. These additions often imply a special compiler, and the code may not be emulated easily on multiple platforms. As a result, special language constructs have not been successful.

Second, due to the programming burden of traditional DSPs, large libraries are typically built up over time. Often more than 1000 functions are provided, including FIR filters, FFTs, convolutions, DCTs, and other computationally intensive kernels. The software burden to generate libraries is high but they can be reused for many applications. With this approach, control code can be programmed in C and the computationally intensive signal processing functions are called through these libraries. This methodology breaks down when the number of lines of C code is large.

Third, when programming in a high-level language such as C, a programmer would often like to take advantage of a specific instruction available in an architecture. Intrinsics were developed because C does not have a mechanism for describing specific instructions. In their rudimentary form, an intrinsic is an asm statement. It has the appearance of a function call in C source code, but is replaced during compilation by a programmer-specified sequence of lower-level instructions [2].

Early intrinsic efforts, like inlined asm statements, inhibited DSP compilers from optimizing code sequences [6]. A DSP C compiler could not know the semantics and side effects of the assembly language constructs. Other solutions which attempted to convey side-effect free instructions have been proposed [2]. These solutions all introduced architecturally dependent modifications to the original C source. Intrinsics which eliminated these barriers were explored in [3]. This technique represented the operation in the intermediate representation of the compiler. With the semantics of each intrinsic well know to the intermediate format, optimizations with the intrinsic functions were easily enabled. This provided speedups of more than 6x. The main detractor of intrinsics is that it moves the assembly language programming burden to the compiler writers. More importantly, each new application may still need a new intrinsic library. This further constrains limited software resources.

3.3 Future DSP Compilation Directions

Future DSP compilers will use a technique called semantic analysis. In semantic analysis, a sophisticated compiler must search for the meaning of a sequence of C language constructs. A programmer writes C code in an architecture independent manner focusing primarily on the function to be implemented. If DSP operations are required, the programmer implements them using standard modulo C arithmetic. The compiler then analyzes the C code, automatically extracts the DSP operations and synthesizes optimized DSP code without the excess operations required to specify DSP arithmetic in C code. This technique has a significant

software productivity gain over intrinsic functions.

Another challenge DSP compiler writers face is parallelism extraction. Early VLIW machines alleviated the burden from the compiler by allowing full orthogonality of instruction selection. Unfortunately this led to code-bloat and excess power consumption. SIMD execution units will be used in modern architectures. However, a vectorizing compiler is required to extract this parallelism. Worse, outer loops must often also be vectorized to allow inner loop vectorization.

3.4 Java

Future 3G wireless systems will make significant use of Java. NTT DoCoMo is already providing Java-based services and may require all 3G systems to support Java[37].

Java is a C++ like programming language designed for general-purpose object-oriented programming[17]. An appeal for the usage of such a language is its "write once, run anywhere" philosophy [18]. This is accomplished by providing a Java Virtual Machine (JVM) interpreter and runtime support for each platform[24].

The language includes a number of useful programming features including programmer defined parallelism support in the form of threads with synchronization, strong typing, garbage collection, classes, inheritance, and dynamic linking. The JVM is a stack-based instruction set designed to efficiently transport programs across the Internet and allow register poor processors to efficiently execute Java bytecode[18]. Instructions are not confined to a fixed length however all of the opcodes in the JVM are 8-bits[24]. This allows for efficient decoding of instructions while not requiring all instructions to be 32-bits or longer.

The Java language supports many of the same basic data representation types as the C language. However, in contrast to C, their values are not implementation dependent[16]. In addition, the Java language also supports `char` which is a 16-bit unsigned Unicode character and a true `boolean` for relational and logical operators. While Java does not allow operations on C-style pointers, it does have the concept of a reference type. These objects are created on a dynamically allocated heap. The distinction is that a reference can not be operated on arithmetically as is often done with C pointers. Operations in the JVM are strongly typed and the 8-bit opcode imposes a constraint of only 256 operations. This results in the tradeoff that nearly all operations are performed as 32-bit integers or IEEE-754 floating point. An interesting Java definition is that the JVM does not indicate overflow or underflow during operations on integer data types[24]. There are also load and store instructions which move values from memory locations to the operand stack in a very RISC-like manner. In addition to standard operations, there is direct support for method invocation, synchronization, exceptions, and arrays. There are two variable length instructions - `tableswitch` and `lookupswitch`.

3.4.1 Java Software Execution

JVM translation designers have used both software and hardware methods to execute Java bytecode. The advantage of software execution is flexibility. The advantage of hardware execution is performance.

Using *interpretation*, a software program emulates the Java Virtual Machine. This requires software to execute multiple instructions for each emulated instruction. This provides cross-platform portability but poses a number of performance issues. While this approach provides for maximum flexibility, the performance achieved can be as low as 5-10% the performance of natively compiled code [19].

Just-in-time (JIT) compilation is an approach where translation is performed just prior to executing the program. JITs have demonstrated 5-10x performance improvement over interpretation[19, 26]. However, the compilation is only resident for the current program invocation. Because they utilize processor resources, the number of optimizations that can be performed prior to execution is restricted. Additionally, multiple instructions are required to implement JVM instructions and there is memory overhead to load the compiler into the runtime system.

Flash compilation is a hybrid approach in that a highly optimizing JIT compiler and a JVM are integrated into a runtime environment[7, 32, 26]. This allows code to be loaded in an already compiled application. The compiler only optimizes loops where a performance gain is likely to be obtained. Performance improvements of 140x interpretive approaches and 13x JIT compilers have been reported.

Off-line compilers, sometimes referred to as way-ahead-of-time compilers, translate Java bytecode to machine code prior to execution. This requires that programs be distributed and installed (e.g. compiled) prior to use. Except for loop information, the Java bytecode contains nearly the same amount of information as the source itself [26]. Therefore, an off-line compiler should be nearly as efficient as a native Java compiler. A restriction on off-line compilers is that all of the class files must be present (e.g. all superclasses) to perform the compilation[1]. Using the Toba compiler, performance improvements nearly twice a standard interpreter have been reported for FFT signal processing functions[12]. The Toba group found performance improvements of 2 to 10 times versus a standard interpreter[27].

Native compilers use Java as a programming language and generate native code directly from the high-level source. A runtime system for linking the Java classes is still required and classes may potentially need to be resolved

each time a method is invoked. Additionally, multiple instructions are required to implement JVM instructions.

3.4.2 Java Hardware Execution

Direct execution is a hardware approach to accelerating Java execution. Sun's *picoJava* implementation directly executes the JVM Instruction Set Architecture (ISA) but incorporates other facilities that improve the system level aspects of Java program execution[25, 30, 34]. Because the JVM does not implement an entire processor, Sun added 115 additional extended bytecodes to the picoJava core. These extended bytecode are not produced by Java compliant compilers. Sun partitions the bytecode into simple instructions which can be directly executed, CISC-like instructions which are implemented in microcode and very complicated instructions which trap. Because a register-file stack cache is used, the picoJava core has access to the top 64 entries in the stack. This allows them to fold (e.g. combine) multiple stack-based operations into one execution packet. On average, Sun found about 28% of instructions executed get folded into other instructions. Researchers at National Chiao Tung University in 1997 found that instruction folding can reduce up to 84% of all stack operations and a 4-foldable Java core could improve overall program speedup by 1.34[5, 33]. Sun states that the picoJava core provides up to 5x performance improvement over JIT compilers [31].

The Delft-Java architecture, designed in 1996, has two logical views: 1) a JVM ISA and 2) a 32-bit RISC-based ISA with both direct and indirect register file access[14]. An important property of Java bytecode is that statically determinable type state enables simple on-the-fly translation of bytecodes into efficient machine code[16]. The Delft-Java processor utilizes this property to dynamically translate Java bytecodes into Delft-Java instructions. Because all bytecodes are stored as pure JVM instructions, JVM bytecode generated by Java compilers executes on this machine without modification. Other Java acceleration techniques used in the Delft-Java processor include a Link Translation Buffer[13], garbage collection[4] and a multithreaded organization[11]. The processor also includes dependency collapsing units that provide facilities similar to instruction folding[35]. Some additional architectural features in the Delft-Java processor which are not directly accessible from JVM bytecode include pointer manipulation, multimedia SIMD instructions, unsigned datatypes, and rounding/saturation modes for DSP algorithms. Speedups of 2x to 3x were realized for implementable machines. This speedup is in addition to the speedup from direct execution.

4 Conclusions

In this paper we have identified some significant challenges facing very high bandwidth 3G cellular systems. The challenge of streaming 2Mbps into a battery operated device places severe constraints on power consumption. To assist this, new hardware techniques using highly programmable paradigms are emerging. The software complexity of SOC systems is also challenging hardware designs. The large signal processing component of communications devices can no longer rely upon assembly language programming. Compiler technology will be crucial to successful system design. A key point of hardware saturating arithmetic units is not only improved performance but the dependency breaking parallel execution capabilities it enables for the compiler. Finally, as Java-based services have already become available for cellular systems, a renewed focus on Java hardware acceleration will become important. Processors that can integrate Java control code and DSP operations will be well positioned for wireless acceptance.

References

[1] A.-R. Adl-Tabatabai, M. Cierniak, G.-Y. Lueh, V. M. Parikh, and J. M. Stichnoth. Fast, effective code generation in a just-in-time Java compiler. In *Proceeding of the ACM SIGPLAN '98 conference on Programming Language Design and Implementation (PLDI'98)*, volume 33, pages 280–290. Association for Computing Machinery, May 1998.

[2] D. Batten, S. Jinturkar, J. Glossner, M. Schulte, and P. D'Arcy. A New Approach to DSP Intrinsic Functions. In *Proceedings of the Hawaii International Conference on System Sciences*, pages 908–918, January 2000.

[3] D. Batten, S. Jinturkar, J. Glossner, M. Schulte, R. Peri, and P. D'Arcy. Interaction Between Optimizations and a New Type of DSP Intrinsic Function. In *Proceedings of the International Conference on Signal Processing Applications and Technology (ICSPAT '99)*, Orlando, Florida, November 1999.

[4] A. Berlea, S. Cotofana, I. Athanasiu, J. Glossner, and S. Vassiliadis. Garbage Collection for the Delft-Java Processor. In *8th IASTED International Conference on Applied Informatics (AI-2000)*, pages 232–238, Innsbruck, Austria, February 2000.

[5] L. C. Chang, L. R. Ton, M. F. Kao, and C. P. Chung. Stack operations folding in Java processors. *IEE Proceedings - Computers and Digital Techniques*, 145(5):333–343, September 1998.

[6] D. Chen, W. Zhao, and H. Ru. Design and implementation issues of intrinsic functions for embedded DSP processors. In *Proceedings of the ACM SIGPLAN International Conference on Signal Processing Applications and Technology (ICSPAT '97)*, pages 505–509, September 1997.

[7] K. Ebcioglu, E. R. Altman, and E. Hokenek. A Java ILP Machine Based on Fast Dynamic Compilation. In *IEEE*

MASCOTS International Workshop on Security and Efficiency Aspects of Java, Eilat, Israel, January 9-10 1997. IEEE Computer Society Press.

[8] European Telecommunication Standards Institute. Digital Cellular Telecommunications System: ANSI-C Code for the GSM Enhanced Full Rate (EFR) Speech Code (GSM 06.53), March 1997. ETS 300 724.

[9] J. Eyre and J. Bier. DSP Processors Hit the Mainstream. *IEEE Computer*, pages 51–59, August 1998.

[10] M. Gagnaire. An Overview of Broadband Access Technologies. In *Proceedings of the IEEE*, volume 85, pages 1958–1972, December 1997.

[11] C. J. Glossner and S. Vassiliadis. The Delft-Java Engine: An Introduction. In *Lecture Notes In Computer Science. Third International Euro-Par Conference (Euro-Par'97 Parallel Processing)*, pages 766–770, Passau, Germany, Aug. 26 - 29 1997. Springer-Verlag.

[12] J. Glossner, J. Thilo, and S. Vassiliadis. Java Signal Processing: FFT's with bytecodes. *Journal of Concurrency and Experience*, 10(11-13):1173–1178, 1998.

[13] J. Glossner and S. Vassiliadis. Delft-Java Link Translation Buffer. In *Proceedings of the 24th EUROMICRO conference*, volume 1, pages 221–228, Vasteras, Sweden, August 25-27 1998.

[14] J. Glossner and S. Vassiliadis. Delft-Java Dynamic Translation. In *Proceedings of the 25th EUROMICRO conference (EUROMICRO '99)*, volume 1, Milan, Italy, September 8-10 1999.

[15] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. Taylor. Piperench: A reconfigurable architecture and compiler. *IEEE Computer*, 33(4), April 2000.

[16] J. Gosling. Java Intermediate Bytecodes. In *ACM SIGPLAN Notices*, pages 111–118, New York, NY, January 1995. Association for Computing Machinery. ACM SIGPLAN Workshop on Intermediate Representations (IR95).

[17] J. Gosling, B. Joy, and G. Steele, editors. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1996.

[18] J. Gosling and H. McGilton. The Java Language Environment: A White Paper. Technical report, Sun Microsystems, Mountain View, California, October 1995. Available from ftp.javasoft.com/docs.

[19] C.-H. A. Hsieh, J. C. Gyllenhaal, and W. mei W. Hwu. Java Bytecode to Native Code Translation: The Caffeine Prototype and Preliminary Results. In *Proceeding of the 29th Annual Internation Symposium on Microarchitecture (MICRO-29)*, pages 90–97, Los Alamitos, CA, USA, December 2-4 1996. IEEE Computer Society Press.

[20] K. Jarvinen et al. GSM Enhanced Full Rate Speech Codec. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, pages 771–774, 1997.

[21] B. Krepp. DSP-Oriented extensions to ANSI C. In *Proceedings of the International Conference on Signal Processing Applications and Technology (ICSPAT '97)*, pages 658–664. DSP Associates, 1997.

[22] P. Lapley. *DSP Processor Fundamentals*. IEEE press, New York, 1997.

[23] K. Leary and W. Waddington. DSP/C: A Standard High Level Language for DSP and Numeric Processing. In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing*, pages 1065–1068. IEEE, 1990.

[24] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1997.

[25] H. McGhan and M. O'Connor. PicoJava: A Direct Execution Engine For Java Bytecode. *IEEE Computer*, 31(10):22–30, October 1998.

[26] G. Muller, B. Moura, F. Bellard, and C. Consel. JIT vs. Offline Compilers: Limits and Benefits of Bytecode Compilation. Technical Report 1063, IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France, December 1996. http://www.irisa.fr.

[27] T. A. Proebsting, G. Townsend, P. Bridges, J. H. Hartman, T. Newsham, and S. A. Watterson. Toba: Java For Applications - A Way Ahead of Time (WAT) Compiler. In *Proceedings Third Conference on Object-Oriented Technologies and Systems (COOTS'97)*, 1997.

[28] M. Saghir, P. Chow, and C. G. Lee. Towards Better DSP Architecture and Compilers. In *Proceedings of the International Conference on Signal Processing Applications and Technology*, pages 658–664, October 1994.

[29] M. J. Schulte, P. I. Balzola, J. Ruan, and J. Glossner. Parallel Saturating Multioperand Adders. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 172–179, 2000.

[30] Sun Microelectronics. picoJava I Microprocessor Core Architecture. Technical Report WPR-0014-01, Sun Microsystems, Mountain View, California, November 1996. Available from http://www.sun.com/sparc/whitepapers/wpr-0014-01.

[31] Sun Microelectronics. Sun Microelectronic's picoJava I Posts Outstanding Performance. Technical Report WPR-0015-01, Sun Microsystems, Mountain View, California, November 1996. Available from http://www.sun.com/sparc/whitepapers/wpr-0015-01.

[32] Sun Microsystems. The Java Hotspot Performance Engine Architecture. Sun Microsystems, 1999. http://java.sun.com/products/hotspot/whitepaper.html.

[33] L.-R. Ton, L.-C. Chang, M.-F. Kao, H.-M. Tseng, S.-S. Shang, R.-L. Ma, D.-C. Wang, and C.-P. Chung. Instruction Folding in Java Processor. In *1997 International Conference on Parallel and Distributed Systems*, pages 138–143, Seoul, Korea, December 12-13 1997. IEEE Computer Society Press.

[34] M. Tremblay and M. O'Connor. picoJava: A Hardware Implementation of the Java Virtual Machine. In *Hotchips Presentation*, 1996.

[35] S. Vassiliadis, J. Phillips, and B. Blanar. Interlock Collapsing ALU's. *IEEE Transactions on Computers*, 42(7):825–839, July 1993.

[36] N. Yadav, M. Schulte, and J. Glossner. Parallel Saturating Fractional Arithmetic Units. In *9th Great Lakes Symposium on VLSI*, pages 214–217, March 1999.

[37] J. Yoshida. Java chip vendors set for cellular skirmish. EE Times, January 30 2001.