When to use a compilation service?*

Jeffrey Palm
Han Lee
Amer Diwan
University of Colorado
{jdp,hanlee,diwan}@cs.colorado.edu

J. Eliot B. Moss
University of Massachusetts
moss@cs.umass.edu

ABSTRACT

Modern handheld computers are certainly capable of running general purpose applications, such as Java virtual machines. However, short battery life rather than computational capability often limits the usefulness of handheld computers. This paper considers how to reduce the energy consumption of Java applications.

Broadly speaking, there are three interleaved steps in running Java programs in a compiled environment: downloading the bytecodes, compiling and possibly optimizing the bytecodes, and running the compiled code. Optimized code typically runs faster than non-optimized code but the optimization process itself may consume significant energy. We consider the possibility of moving compilation (optimizing or non-optimizing) to a tethered server. We demonstrate that there is a significant benefit to moving compilation to a server (up to 67% reduction in energy for a realistic handheld configuration). We also demonstrate that there is no single best compilation strategy for all methods.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors–Compilers, Optimization

General Terms

Experimentation, Languages, Measurement, Performance

Keywords

Distributed compilation, Java, Energy efficient compilation

1. INTRODUCTION

Modern handheld computers are capable computing devices with fast processors and large memories. They are extensible, and run general purpose operating systems such as Linux and WinCE. For example, the relatively new Compaq iPAQ 3835 has a 206MHz StrongArm processor, 64MB of RAM, an expansion slot (which can take PCMCIA cards) [7] and runs WinCE and Linux. In contrast, the older Palm Vx has a 20MHz processor, 8MB of RAM, limited expandability [9], and runs Palm OS, a special purpose operating system. While the increased capability of modern handhelds allows them to run powerful and general applications (such as the Java virtual machine), their short battery life severely restricts their usefulness. In this paper we examine how to reduce the energy consumption of Java applications. Our Java infrastructure uses compilation only, and does not support interpretation. We defer study of systems that use interpretation (perhaps in addition to compilation) to future work.

Broadly speaking, there are three *interleaved* steps in running Java programs in a compiled environment: downloading the bytecodes, compiling and possibly optimizing the bytecodes, and running the compiled code. These steps are demand driven. When a program references a class for the first time, the Java system downloads the class. When a program calls a method for the first time the Java system compiles the method, possibly with optimization. For this research we extend the Java model by assuming that a handheld can download not only Java bytecodes but also compiled or optimized versions of the bytecodes. There are therefore four main possibilities: (i) download bytecodes and compile them locally *without* optimizations; (ii) download bytecodes and compile them locally *with* optimizations; (iii) download compiled but unoptimized code; and (iv) download optimized code. For options (iii) and (iv) the handheld may have to wait in idle mode for code to be compiled or optimized on a server. These configurations offer different tradeoffs between compilation time, download effort, and execution time. Bytecodes will probably be more compact than compiled code and will thus incur lower download energy costs and download time. Optimized code will probably run faster than compiled code but it will take more energy to optimize the code or to wait for the code to be optimized.

This paper evaluates the above-mentioned configurations with respect to their energy costs. We conducted these experiments in the Jikes RVM Java System [1] using the SPECjvm98 benchmarks. We use an analytical model of energy with parameters drawn from real hardware (such as the Itsy [13] or Compaq WL100 wireless card [8]). We also vary our parameters and input sizes to increase the generality of our results.

Our results demonstrate that, at least for the optimizations in Jikes RVM, optimizing methods on a handheld is extremely costly with respect to energy. Even with relatively long benchmark runs, we are unable to recover the optimization cost. However, using a server to optimize code is beneficial, reducing end-to-end energy

*This work is supported by NSF ITR grant CCR-0085792, an NSF CAREER award, an IBM faculty partnership award, and an equipment gift from Intel. Any opinions, findings and conclusions or recommendations expressed in this material are the authors' and do not necessarily reflect those of the sponsors.

LCTES'02–SCOPES'02, June 19-21, 2002, Berlin, Germany.

consumption by up to 67%. We find that it is important to avoid *waiting* for the server to optimize; the wait is often so long that it negates the benefit of using a server, particularly for the shorter-running methods and benchmarks.

The remainder of this paper is organized as follows. Section 2 presents background and motivation for the paper. Section 3 describes and discusses our experimental methodology. Section 4 presents the results. Sections 5 and 6 review prior work in the area and suggest directions for future work. Finally, Section 7 concludes the paper.

2. BACKGROUND

Users may download, compile, and run Java applications by simply clicking on links in their web browser. In the traditional Java model, the web browser downloads applications in bytecode format [2], and a local Java Virtual Machine (JVM) interprets the bytecodes or compiles them and runs them. As the program runs, it may download more bytecodes. For security, the JVM checks (validates) the bytecodes to ensure that they do not violate certain safety properties.[1]

Most high-performance JVMs today, such as HotSpot [18] or Jikes RVM [1], employ compilation rather than or in addition to interpretation, since compilation (and its corresponding optimizations) leads to faster code. Compilation and optimizations, however, consume both time and energy. In this work we consider how to reduce the energy overhead of compilation and optimization while retaining their benefits.

We consider an alternate model of Java execution where one can download either bytecodes or compiled (and possibly optimized) code. By downloading compiled code, our model borrows from that of Sirer et al. [22]. In that model, each organization has one or a few local proxy servers that satisfy all web accesses for the organization. These proxy servers provide many services (such as optimizations) and transform or translate the code as it passes through them to the local client. Using a server to do expensive compilation and optimization is an example of "cyber-foraging" [21].

We now define two terms that we use throughout the paper. A *server* is a machine that is connected to a power outlet (i.e., it does not rely on batteries for power) and can provide services such as code optimization. A *client* is a handheld computer running on batteries and has wireless capability with which it can communicate with servers.

3. METHODOLOGY

We now describe our measurement infrastructure, the different configurations that we evaluate, measurement techniques, and the assumptions that we make.

3.1 Infrastructure

We conduct our experiments in the Jikes RVM infrastructure from IBM Research [5]. Jikes RVM contains an aggressively optimizing compiler that performs many leading-edge optimizations for object-oriented programs, including method inlining, method resolution, and many scalar optimizations. Jikes RVM compiles code on demand: it loads classes when the program uses them for the first time and compiles methods when the program calls them for the first time. One can build the system so that it always uses the baseline (non-optimizing) compiler or always uses the optimizing compiler. Jikes RVM also contains an adaptive optimizer [3] that tracks hot spots in code and optimizes or reoptimizes them; for our experiments, we did not use the adaptive optimizer. Unless otherwise stated, by *optimizations* we refer to all normal optimizations in Jikes RVM *except for method inlining*.

[1]Jikes RVM, because it is targeted for research use rather than commercial deployment, does not perform bytecode validation.

3.2 Configurations

We consider four main configurations:

1. **localBaseline**. Download bytecode on client and compile locally without optimizations.
2. **localOpt**. Download bytecodes on client and compile locally with optimizations (-O2).
3. **serverBaseline**. Download unoptimized compiled code from server.
4. **serverOpt**. Download optimized (-O2) compiled code from server.

The energy consumption of a given configuration, C, has several components:

1. $\mathbf{E_{download}(C)}$. Energy consumed by the wireless card while downloading code.
2. $\mathbf{E_{wait\text{-}download}(C)}$. Energy consumed by the client while waiting for code to download.
3. $\mathbf{E_{wait\text{-}compile}(C)}$. Energy consumed while waiting for code to compile or optimize on the server.
4. $\mathbf{E_{compile}(C)}$. Energy for compiling or optimizing on the client.
5. $\mathbf{E_{run}(C)}$. Energy for running the compiled application on the client.

Table 1 give the equations we use to compute the energy consumption of our four configurations. One or more of the above energy components may not exist for a given configuration. For example, $E_{compile}(C)$ only exists for the localBaseline and the localOpt configurations. Also, a given component may contribute differently to different configurations. For example, the E_{run} component of the localOpt and serverOpt configurations may be the same but hopefully much lower than the E_{run} component of the localBaseline and serverBaseline configurations. Note that we are assuming that servers do not run on batteries and thus their energy consumption is not important to this study.

3.3 Computing the components

Prior work has noted that the energy consumption of a program for the most part varies linearly with execution time [25, 26]. In our own experiments we found the rate of energy consumption on an actual Itsy pocket computer[2] [10] averaged 2.22 Watts with a standard deviation of 0.52 Watts. In other words, to a reasonable approximation we could compute the energy consumption of a program on an Itsy by multiplying its execution time by 2.22. Note, of course, that compiler and run-time system optimizations that target energy consumption could cause a program's rate of energy consumption to deviate significantly from 2.22 Watts. However, since we are not considering any such optimizations in this study we felt that it was appropriate to use execution time as an indicator of energy consumption.

We instrumented Jikes RVM running on a 1.4 GHz Pentium 4 desktop to measure compile time and run time of each method, the

[2]The Itsy pocket computer is similar to a Compaq iPAQ pocket PC.

Configuration	Equation
localBaseline	$E_{download}(bc) + E_{wait\text{-}download}(bc) + E_{compile}(base) + \quad + E_{run}(base)$
localOpt	$E_{download}(bc) + E_{wait\text{-}download}(bc) + E_{compile}(opt) + \quad + E_{run}(opt)$
serverBaseline	$E_{download}(base) + E_{wait\text{-}download}(base) + \quad + E_{wait\text{-}compile}(base) + E_{run}(base)$
serverOpt	$E_{download}(opt) + E_{wait\text{-}download}(opt) + \quad + E_{wait\text{-}compile}(opt) + E_{run}(opt)$

Table 1: Equations for computing components of energy

Constant	Value used
Download cost per byte	444 nanoJoules/byte
Download speed	11 Mbits/sec
Handheld power	2 Watts
Idle ratio	0.5
Slowdown ratio	7

Table 2: Constants used in our experiments

Name	Description
check	Tests various features of the JVM
compress	Modified Lempel-Ziv method
jess	Java Expert Shell
db	Database operations on a memory resident database
javac	Java compiler from the JDK 1.0.2
mpegaudio	Decompresses audio files
jack	Parser generator, earlier version of JavaCC

Table 3: Benchmark descriptions.

size of the method bytecodes, and the size of the compiled methods. The run time of a method includes only the time spent in the method and not in its callees. We assume that the handheld is also Pentium-based and runs a constant factor slower than the desktop.

Table 2 gives the constants we used to compute energy from our time and size measurements. We obtained the download cost per byte and transmission speed from the published numbers for the Compaq WL110 wireless card [8]. While a handheld is receiving data, it consumes energy (at the "handheld power" rate in Table 2) in addition to the energy consumed by the card. The "Handheld power" is the power consumed by the handheld when not in idle mode, i.e., what we call "Peak power". We obtained this number from our experiments with the Itsy [10]. The idle ratio is the power consumed when in idle mode (e.g., waiting for the server to compile code) divided by the peak power consumption. The "Slowdown ratio" is the speed of the handheld compared to the workstation on which we measured the times. We computed this slowdown by timing two applications on our measurement workstation and on a set-top box using a configuration and (processor) similar to the Itsy.

Section 3.5 discusses the simplifications embodied in these constants.

3.4 Benchmarks

Table 3 describes the benchmark programs. All benchmarks are taken from the SPECjvm98 suite and use the large inputs (100). These are commonly used and non-synthetic benchmark programs. We omit one SPECjvm98 benchmark (*mtrt*) since our measurement infrastructure cannot yet correctly handle multithreaded programs.

Table 4 gives the size in bytes of the bytecodes, baseline (i.e., compilation without optimizations) code, optimized code, and optimized code with all optimizations including inlining enabled for each benchmark program. Note that the bytecode sizes are much

Benchmark	Bytecode	Base.	Opt.	Opt. + inl.
check	20.9	111.4	129.8	162.8
compress	15.2	71.3	94.4	102.3
jess	35.5	190.5	264.6	336.7
db	16.6	78.1	106.4	134.7
javac	35.9	172.9	234.8	308.5
mpegaudio	28.0	147.2	165.1	208.9
jack	38.8	227.4	240.3	302.5
G. mean ratio	1.00	5.17	6.40	7.95

Table 4: Size in KB of the bytecode, and compiled and optimized (for x86) machine code

smaller than the compiled code sizes. Also, optimizations, and in particular inlining, further increase the size of the compiled code. For each configuration the *G. mean ratio* is the geometric mean of the code size normalized to the bytecode size.

Table 5 gives the time (in seconds) to compile, optimize, and run the benchmarks. The execution times in this table are the total time for running the benchmarks and thus they also include time spent in standard libraries. As expected, the optimization time is two orders of magnitude higher than simple compilation. Also, once optimized, the code runs significantly faster particularly with inlining enabled. The *G. mean ratio* gives the geometric mean of the compile or execution time normalized to the compile or execution time of the baseline configuration.

3.5 Simplifications

The most precise way to conduct this study is to implement the distributed compilation framework on both a handheld and a server platform and use either hardware or detailed energy simulations to measure the energy costs. Since the focus of this study was to understand whether or not there is any benefit to using a server to compile and optimize on behalf of the handheld, it made little sense to build the full hardware and software infrastructure. Thus, our study is less precise and makes some simplifications.

- Energy consumption is proportional to execution time. While this is largely true, we do notice that power consumption can vary by up to 25% in our experiments with an actual handheld. To account for this variation and also to account for variation across handhelds, we report on how our results would change with a range of power consumption.
- The handheld consumes 50% of its peak power when idle. Other researchers have made similar assumptions [4]. Flinn et al. report that an Itsy comsumes approximately 18% of energy when idle compared to when it is in a tight compute loop. To account for this variation in different handhelds, we report how our results change when we consider different factors for idle energy over active energy.
- Transmission energy can be approximated by multiplying the number of transmitted bytes with the average energy to trans-

Benchmark	Compile time			Execution time		
	Base.	Opt.	Opt. + inl.	Base.	Opt.	Opt. + inl.
check	0.1	15.3	24.7	2.3	1.7	0.4
compress	0.2	11.6	19.8	54.1	21.6	18.0
jess	0.1	24.2	39.0	27.6	20.7	12.3
db	0.1	12.7	21.8	46.7	31.3	29.1
javac	0.2	24.1	37.9	13.6	12.2	10.2
mpegaudio	0.2	20.8	27.2	53.2	42.0	30.2
jack	0.2	26.0	43.5	28.8	22.8	16.1
G. Mean ratio	1.0	137.9	219.9	1.0	0.7	0.4

Table 5: Time (in seconds) for compiling and running the benchmarks on a 1.4GHz Pentium 4 workstation

mit one byte. This simplification ignores acknowledgement or retry packets. In our experiments we discovered that the download energy was often one or two orders of magnitude smaller than compilation or execution energy; thus, adding the energy for the acknowledgement or retry packets will probably not change our results.

- Our instrumentation does not perturb compiler optimizations. To understand the impact of our instrumentation on compiler optimizations, we compared the overall program speedup obtained using our per-method instrumentation to the speedup obtained by timing the full runs (without our instrumentation). We found these speedups to be comparable (though not identical since we do not instrument certain core libraries which would be measured in the timing of the full runs).

- The handheld already has appropriate versions of the VM code (such as garbage collector). In other words, we assume that the core VM code does need to be downloaded or compiled. This is a reasonable assumption since the VM core services are used by all programs and thus should be available in pre-compiled form.

4. RESULTS

Section 4.1 compares the energy consumption of our different configurations. Section 4.2 presents similar numbers except broken down by Java methods. Section 4.3 breaks down energy consumption into its components. Section 4.4 considers our results when we use smaller inputs for our benchmarks. Section 4.5 reports the effect of varying our parameters. Finally Section 4.6 summarizes the results.

4.1 Same configuration for all methods

Table 6 compares the energy costs of our different configurations normalized to the energy cost of localBaseline, including the geometric mean of each column at the bottom. The "NoIdle" configurations assume that the server optimizes all methods before the client needs them, and thus the client does not have to wait (idle) while the server optimizes the code. From the table we see that moving non-optimizing compilation to the server has little benefit over localBaseline. In other words, the slight savings by moving non-optimizing compilation onto a server balances the greater cost of downloading compiled code instead of the smaller bytecodes.

For the optimizing configurations, localOpt and localOpt with inlining consume far more energy than localBaseline except for compress. In other words, the cost of locally optimizing and inlining code is so great that for most benchmarks it overwhelms the benefit of optimizing the code. From Table 5 we see that the feasibility of localOpt depends greatly on how the execution time of the benchmark compared to its compilation time.

The serverOptNoIdle configuration consumes the least energy overall for our benchmark programs. However, serverOptNoIdle may not be realistic since it assumes that the server is able to compile and optimize all the code before the handheld needs it. The serverOpt configuration also performs well but as expected, not as well as serverOptNoIdle. For configurations without inlining, serverOpt actually performs slightly worse than localBaseline for two benchmark programs.

To summarize, our data tells us that for realistic configurations, optimizing on the server is often, but not always, a good idea. If we can predict in advance which methods will be needed in the future of the computation (i.e., approach the *NoIdle* configurations) optimizing on the server is beneficial for all programs.

4.2 Different configurations for each method

Section 4.1 evaluates configurations based on their overall energy consumption. However, it does not explore whether it is worthwhile to use a *mixture* of configurations for a benchmark, for example, using localBaseline for some methods and serverOpt for others.

Table 7 presents the energy consumption of an "Optimal" configuration. This configuration picks the best compilation strategy for each method. For comparison, the table also presents results for the two configurations that optimize (but do not inline) on the server. The results for serverOptNoIdle and serverOpt in Tables 6 and 7 are different because Table 6 uses end-to-end program execution time while Table 7 uses per-method execution times (which are unavailable for certain core classes, such as the garbage collector).

The reason for omitting inlining from Table 7 is practical: we cannot determine the time spent in a method (but not its callees) in the presence of inlining since inlining (and subsequent optimizations) can interleave the instructions of callers and callees. For four of the benchmarks we see that "Optimal" performs better than serverOptNoIdle. In other words, using the same configuration (e.g., serverOptNoIdle) for all methods is suboptimal. To get the best energy consumption, we must decide what kind of compilation to use on a per-method basis.

Figures 1 and 2 show the energy each method consumes, as a fraction of the energy it consumes with localBaseline. The energy consumption of a method includes costs of downloading, compiling (or waiting for compilation on the server), and running the method (Table 1). The graphs are cumulative along the horizontal axis. In other words, a point (x,y) says that x% of the methods consume y or less energy compared to what the method consumes with localBaseline. The methods are sorted in order of increasing ratio of energy consumption compared with localBaseline.

Each graph has one curve for each configuration (localOpt, serverOpt, serverBaseline) plus two additional curves for the "NoIdle" configurations. In the legends, *IdleR* is the idle ratio and *Power* is the

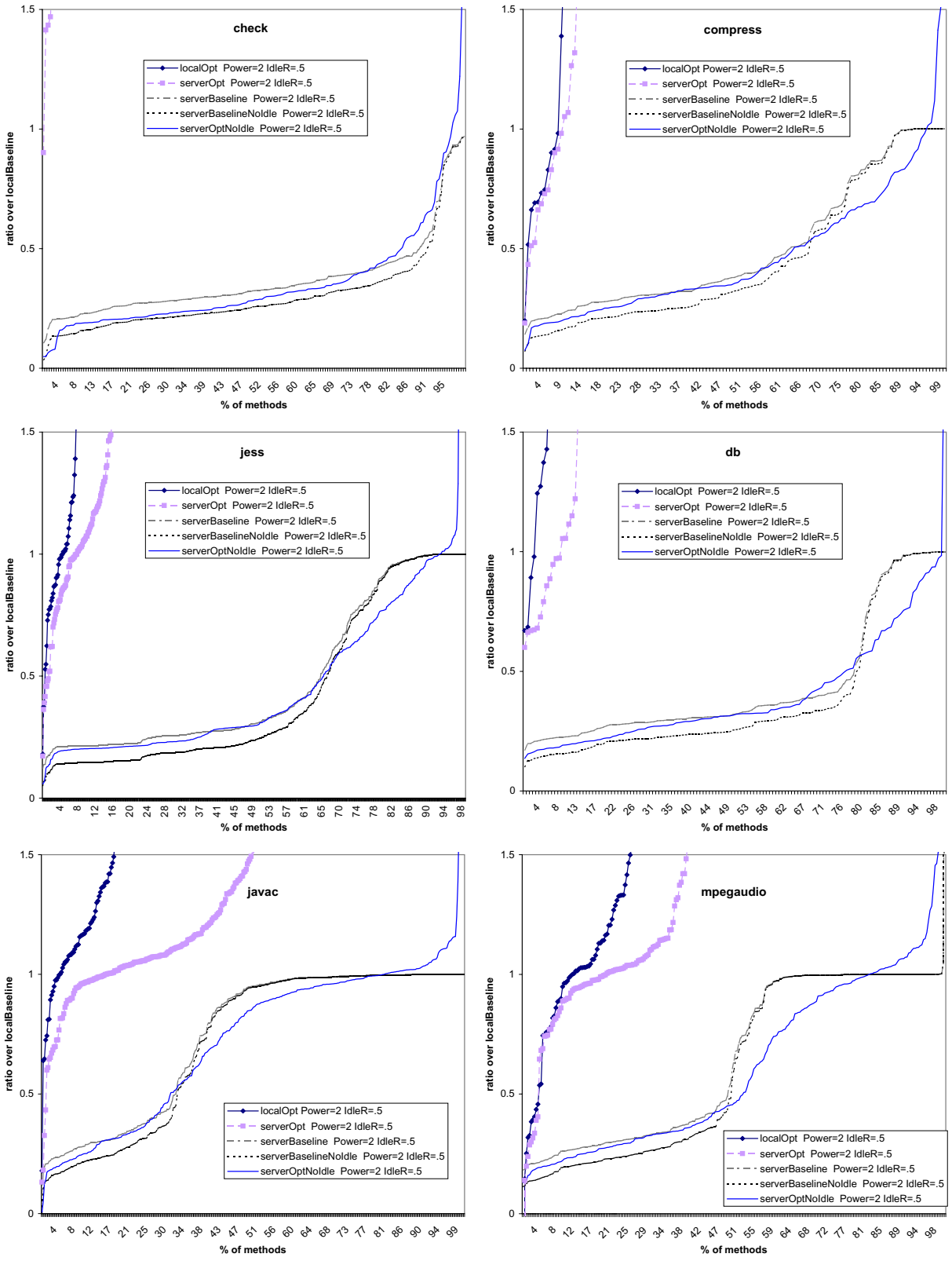


Figure 1: Cumulative energy consumption normalized to localBaseline

Benchmark	serverBaselineNoIdle	serverBaseline	Without inlining			With inlining		
			localOpt	serverOptNoIdle	serverOpt	localOpt	serverOptNoIdle	serverOpt
check	0.97	0.97	7.01	0.69	1.14	10.41	0.17	0.90
compress	1.00	1.00	0.24	0.03	0.04	0.70	0.33	0.33
jess	1.00	1.00	1.62	0.75	0.81	1.85	0.44	0.44
db	1.00	1.00	0.94	0.67	0.69	1.09	0.62	0.62
javac	0.99	0.99	2.64	0.89	1.02	3.50	0.74	0.74
mpegaudio	1.00	1.00	1.18	0.79	0.81	1.08	0.57	0.57
jack	1.00	1.00	1.69	0.79	0.85	2.06	0.56	0.56
Geo. Mean	0.99	0.99	1.45	0.48	0.57	1.97	0.45	0.57

Table 6: Energy costs of different configurations compared to localBaseline

Benchmark	serverOptNoIdle	serverOpt	Optimal
check	0.51	11.70	0.43
compress	0.76	0.76	0.76
jess	0.86	0.87	0.83
db	0.69	0.70	0.69
javac	0.88	0.93	0.87
mpegaudio	0.55	0.56	0.54
jack	0.61	0.65	0.61
Geo. Mean	0.68	1.09	0.66

Table 7: Energy costs for optimal configuration normalized to energy cost of localBaseline

handheld energy consumption per unit time. For these figures we keep *IdleR* and *Power* fixed. In Section 4.5 we vary these parameters and study their impact on the results. Note that the "opt" configurations include all optimizations except for inlining. To make the curves easier to identify (in this and later graphs), the order in the legend (bottom to top) matches as much as possible to the order of the curves (bottom to top).

From Figures 1 and 2 we see that the curves for serverOptNoIdle, serverBaseline, and serverBaselineNoIdle overlap significantly. However, for a significant number of methods serverOptNoIdle is the best configuration. serverOpt, on the other hand, performs poorly, and is worse than localBaseline (i.e., its curve is above 1) for the majority of the methods. These results seem to contradict the results in Table 6 which shows that serverOpt is significantly better than serverBaseline and serverBaselineNoIdle, while worse than serverOptNoIdle. The reason for this apparent contradiction is that Figures 1 and 2 weigh all methods equally, while Table 6 weighs methods by the amount of time spent in the method. Thus, even though serverOpt performs much worse than serverBaseline for most methods, it performs better than serverBaseline for the handful of methods that contribute most to execution time (and thus energy consumption).

The serverBaselineNoIdle configuration is also often better than localBaseline, but curiously, the serverBaselineNoIdle curves all flatten out at 1. Here is why. The main benefit of serverBaselineNoIdle over localBaseline is that the handheld does not need to compile the code.[3] If the program spends little time in the method, then the compilation cost of the method will dominate its energy consumption. If on the other hand, the program spends significant time in the method, then its compilation cost will be amortized over a longer execution time and thus, the execution time of the method will dominate its energy consumption. Based on these observations, serverBaselineNoIdle is better than localBaseline for methods with a low running time, but serverBaselineNoIdle and localBaseline perform asymptotically the same for methods with longer running times. Hence the ratio curves approach 1.

[3]We found the download energy to be much smaller than energy for baseline compilation.

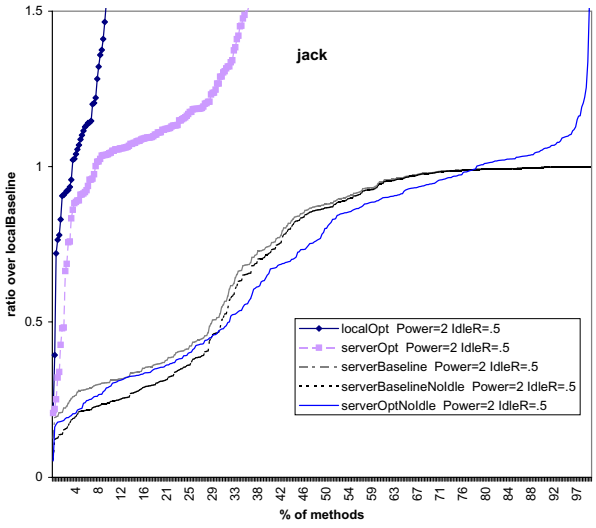


Figure 2: Figure 1 continued

While serverBaseline and serverBaselineNoIdle have almost identical performance, serverOpt and serverOptNoIdle have very different energy performance. This occurs because the time to compile without optimizations is insignificant (and thus we consume little idle energy) compared to the time to optimize (Table 5).

Finally, localOpt performs poorly, yielding better energy consumption than localBaseline for only a few percent of the methods. From Table 6 we see that localOpt consumes much more overall energy than localBaseline.

To summarize, for the majority of methods, it makes sense to optimize on the server especially if we can avoid waiting for the server to optimize. In other words, serverOptNoIdle is the configuration of choice. However, there are still some methods for which localBaseline performs better than serverOptNoIdle; we are trying to find ways to identify these methods in our future work (Section 6).

4.3 Breakdown of energy consumption

We now break down the energy consumption of methods into their components. Figures 3 (a), (b), and (c) break down the energy consumption of the localBaseline, serverOptNoIdle, and serverOpt configurations for one benchmark program, *javac*. The graphs for other benchmarks are similar so we omit them. These graphs plot

the energy consumption of a method (in Joules) versus the execution time of the method. By execution time of a method we mean the total time spent in the method (but not its callees) during a run of the program. That is, we are summing over all invocations of the method, not just a single invocation. The vertical axis (energy) uses a log scale, and the energy components are stacked (in the same order as the legends) one on top of another.

From the localBaseline breakdown graph (Figure 3 (a)) we see that for methods with low execution time energy consumption is dominated by the energy costs of compiling the code (the transmission energy is negligible). For methods with relatively high execution time, the method execution component dominates the energy consumption. Figure 3(c) (serverOpt) is similar; however, instead of local compilation time, it has the overhead of waiting for the server to optimize.

From the serverOptNoIdle breakdown graph (Figure 3 (b)) we see that except for methods with very low execution time, the execution dominates the energy consumption.

We also note that transmission energy in the configurations that optimize on the server are much higher than the transmission energy of localBaseline (which is nearly invisible in Figure 3(a)). This is because optimized code is much larger than bytecodes (Table 4).

These results suggest that for methods with low execution time serverOptNoIdle has an advantage over localBaseline, since serverOptNoIdle incurs no compilation energy costs. On the flip side, for these methods, localBaseline has an advantage over serverOpt, since serverOpt has to consume energy waiting for the server to optimize. Moreover, the idle energy is not recovered afterwards in the (hopefully shorter) execution time of the optimized code.

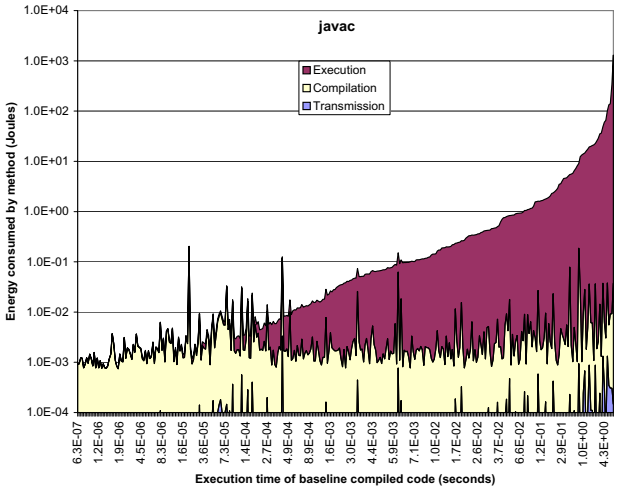
For methods with long execution time, the main advantage of serverOptNoIdle over localBaseline is faster execution time. For methods with short execution time, serverOptNoIdle also eliminates the compilation overhead for localBaseline which is a significant component of energy for these methods. Thus, serverOptNoIdle will have better energy consumption than localBaseline for long-running methods, but the improvement will not be as dramatic as for short-running methods. Finally, serverOpt will perform better for long-running methods than for short-running methods, because for long-running methods, the idle time is amortized over longer a longer execution.

Figure 4 confirms the above argument by presenting the energy consumption of serverOpt and serverOptNoIdle configurations for one benchmark program (*javac*). The graphs for the other benchmarks are similar. Each point in this graph corresponds to the energy consumption of one method in either the serverOpt or serverOptNoIdle configurations. The energy consumption in the graphs is normalized to the energy consumption of localBaseline. From this graph we see that for the short-running methods, serverOpt is much worse than localBaseline and serverOptNoIdle is much better than localBaseline. For long-running methods, both serverOpt and serverOptNoIdle usually perform as well or better than localBaseline, with serverOpt performing slightly worse than serverOptNoIdle.

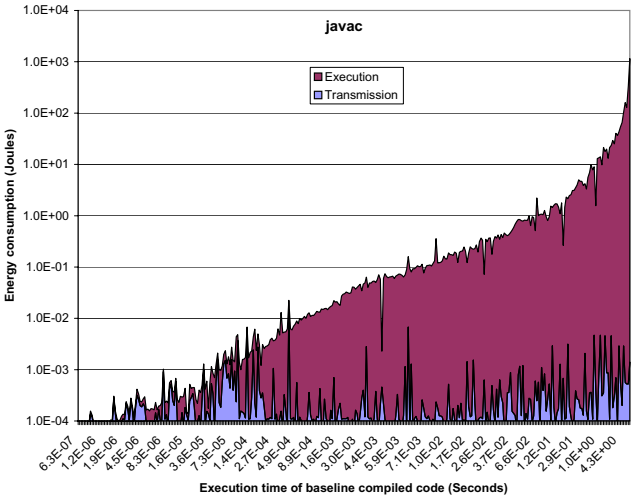
4.4 Smaller input sizes

Figure 5 presents results for *javac* using smaller inputs (input 1 from SPECjvm98). This graph should be compared to the graph for *javac* in Figure 1. We omit the graphs for the other benchmarks since they are similar.

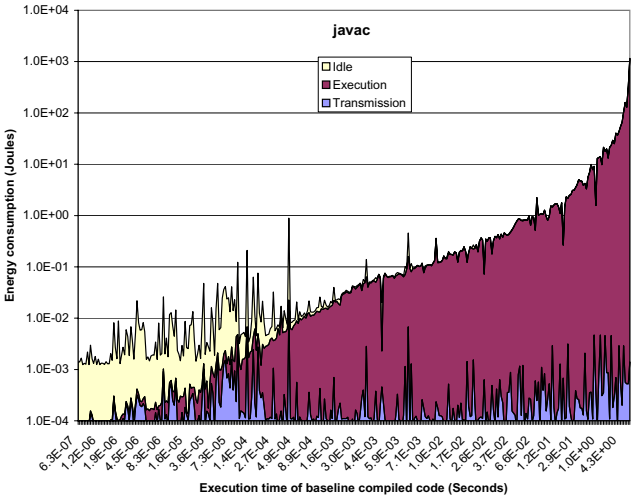
From Figure 5 we see that serverOptNoIdle, serverBaseline, and serverBaselineNoIdle continue to perform well compared to localBaseline. However, unlike the *javac* graph in Figure 1, serverOptNoIdle performs slightly worse than serverBaseline and serverBase-



(a) Energy consumption breakdown of localBaseline



(b) Energy consumption breakdown of serverOptNoIdle



(c) Energy consumption breakdown of serverOpt

Figure 3: Breakdown of energy consumption

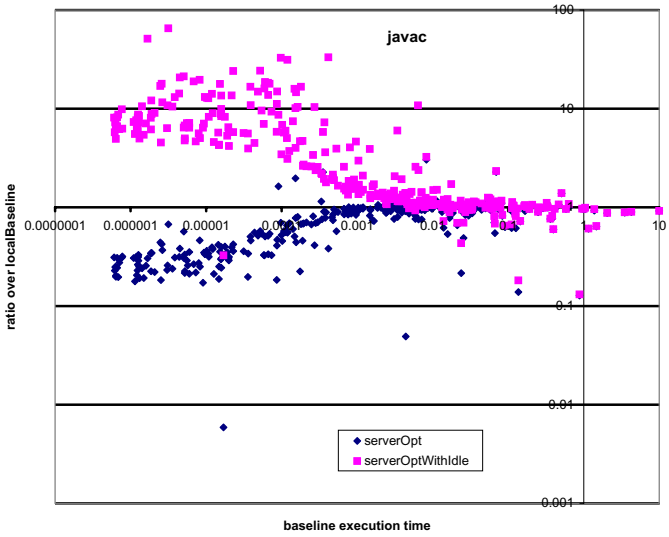


Figure 4: Energy consumption sorted by execution time of baseline-compiled methods.

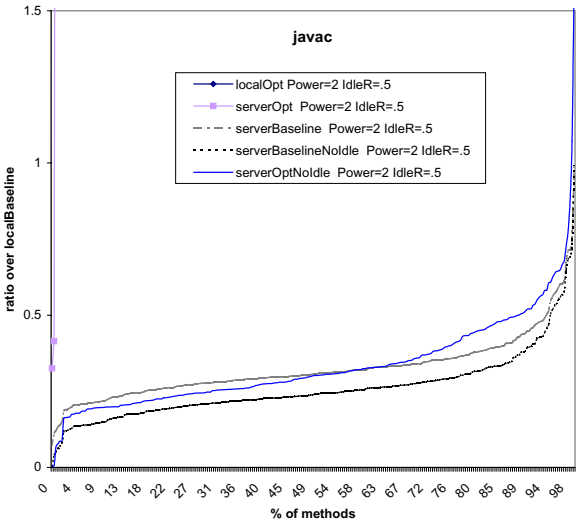


Figure 5: Cumulative energy consumption normalized to localBaseline (using small inputs)

lineNoIdle. Also, with the small inputs serverOpt performs much worse than localBaseline and localOpt performs so poorly that its curve is not even visible in Figure 5. The reason for these differences between the energy consumption of the small and large inputs is that with the smaller inputs we are frequently unable to recover the energy cost of the optimizations (which is the same for both small and large inputs) and even transmission.

4.5 Sensitivity analysis

First, we observe that our results are not very sensitive to the downloading power consumption rate. Figure 3 shows clearly that energy for downloading is not a big factor overall, so changing the download power by a factor of 2 or more will not significantly change the overall outcomes.

We now consider the idle ratio, IdleR, and the peak power consumption, Power. So far, we have fixed the rate of energy consumption of the handheld when active and when in idle mode to 2 and 1 Watts respectively. We now vary both parameters to approximate a range of design choices in the handheld: from more aggressive handhelds that consume energy at a higher rate to severely resource-limited handhelds such as ones that may be found in highly-portable devices.

Figure 6(a) shows how the energy consumption of serverOptNoIdle changes when handheld power varies from 0.2 to 4 Watts. (These graphs follow the same format as the graphs in Figures 1 and 2.) We hold the idle ratio constant at 0.5 (i.e., half of the handheld power) for these graphs. We also assume that the speed of the handheld does not change with changing energy (i.e., the reduced energy consumption comes from not a slower device but from a more efficiently designed device or one with a less sophisticated display). We present results for only one benchmark, *javac*, since the other benchmarks yield similar graphs.

From Figure 6(a) we see that decreasing the handheld power consumption has relatively little impact on the shape of the serverOptNoIdle curve. The 0.2 curve is the one that changes the most. Lowering the handheld power consumption to 0.2 Watts lowers the energy needed to run the program and thus the benefit of optimizations in reducing execution time becomes less important.

Figure 6(b) shows how the energy consumption of serverOpt changes when the idle ratio changes from 0.5 to 0.1 (i.e., half to a tenth of the handheld power consumption). These graphs follow the same format as the graphs in Figure 1 and 2). We hold the handheld power constant at 2 Watts for these graphs. We present results for only one benchmark, *javac*, since the other benchmarks yield similar graphs.

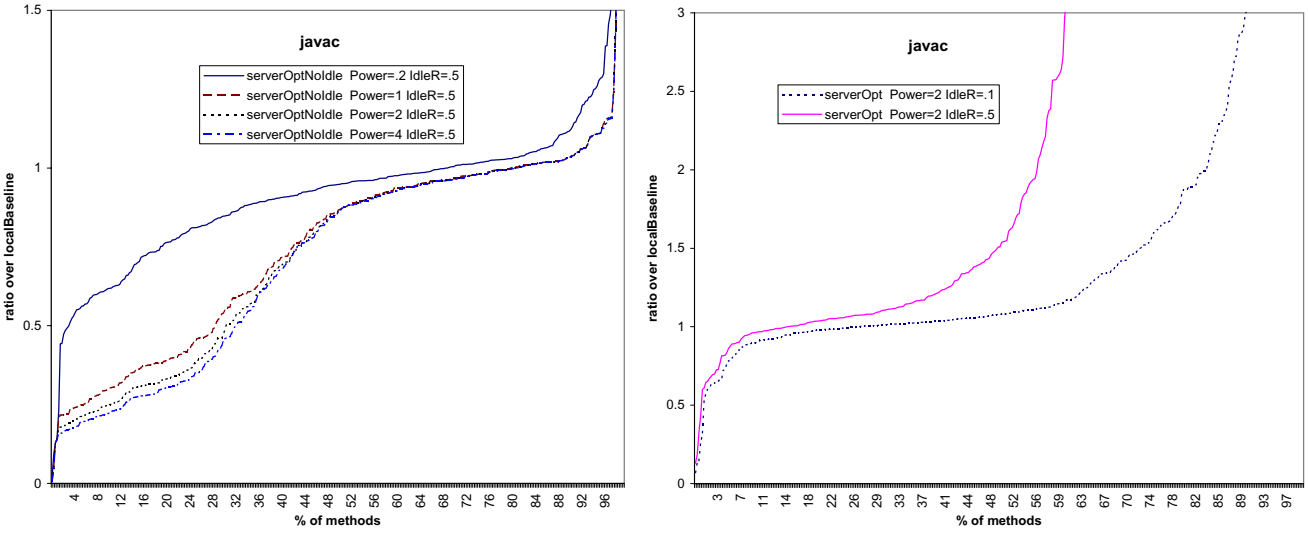
From Figure 6(b) we see that while lowering the idle ratio has a great impact on the shape of the curve, both the 0.5 and 0.1 curves cross localBaseline at about the same point. In other words, lowering the idle ratio does not affect the number of methods for which serverOpt is more attractive than localBaseline.

4.6 Summary of results

Our results indicate that a sufficiently smart implementation of distributed compilation could significantly improve the energy consumption of Java programs running on a handheld. However, to get the maximum benefit, we need to be either selective about which methods we optimize or use precompilation to avoid waiting (and wasting energy) while the server is optimizing code.

5. RELATED WORK

The research of Rudenko et al. [19, 20] is the most similar to ours. They report the energy benefits of migrating tasks onto a tethered server from a laptop. One of the tasks they consider is



(a) Changing energy consumption

(b) Changing idle energy consumption

Figure 6: Changing energy parameters.

incremental compilation.[4] They measure energy consumption using the APM metrics to indicate the energy remaining in the battery. They demonstrate that there is often a significant benefit to migrating compilation to a server. Our work differs from theirs as follows. First, we consider both optimizing and non-optimizing compilation. Second, we consider energy consumption on a handheld as opposed to a laptop. Compared to laptops, handhelds have smaller and lower resolution screens and no disk (but flash memory), therefore the energy behavior of a handheld will be different from a laptop. Third, we conduct our experiments for Java programs and consider the possibility of deciding on a per-method basis how and where to compile them.

Flinn et al. [12] describe a framework that automatically downloads tasks to a wired server based on information provided by the application and past profiles. Their work also incorporates a notion of "fidelity"; for example, their system may decide based on the environment to use either the full or a short vocabulary for a speech recognition system. Our work is in a sense a detailed evaluation of one application of their framework.

Kremer et al. [16, 17] describe a framework for migrating the execution of applications on to a server. The server periodically (as determined by the compiler) sends checkpoints to the handheld; if the server dies or is disconnected, the handheld can continue execution from the last checkpoint. These ideas are useful in building a distributed compilation system.

Teodorescu and Pandey [23] describe a Java system that is distributed across servers and resource-limited devices. The resource-limited devices run minimal kernels that download parts of the runtime system on demand. Like our work, the granularity of transferring code is a method. However, unlike our work, all compilation is done on the server. Teodorescu and Pandey do not investigate the energy costs of distributing the tasks of a Java virtual machine.

Sirer et al. [22] consider how to distribute the tasks of a Java virtual machine between a personal computer and a proxy server. By putting tasks such as bytecode verification on a server, they expect to enhance the reliability and stability of a system. For example, a system administrator would need to install many security patches on only a small number of servers rather than on every computer in the organization. Since the proxy servers are behind firewalls and relatively secure, a client in the organization can trust the server. We follow the model of Sirer at al. for our work. However, the motivations behind our work and theirs are different: they are concerned with security and performance while we are concerned with energy consumption.

Two prior projects have investigated the energy consumption of a Java virtual machine on a handheld. Farkas et al. [11] report on the energy consumption of a Java virtual machine running on a StrongARM-based Itsy [27]. Farkas et al. conduct their measurements using special hardware. Chen et al. [6] describe the effect of variations in a mark-and-sweep garbage collection algorithm in a severely resource limited environment. Chen's environment is a microSPARC-IIep based 100MHz system with 128 KB of ROM and no caches. Both above-mentioned works assume that all compilation or interpretation happen on the handheld and do not consider the possibility of migrating the compilation to a server.

Finally, there has been much work on compiler optimizations for reducing energy consumption of programs [24, 14, 15].

6. FUTURE WORK

The research presented here points us to several questions worthy of further study:

- How to predict which methods to optimize, how to optimize them, and where to optimize them? In this paper we only considered two levels of optimizations: with and without inlining. However, different optimizations have very different properties in terms of how they affect code size, how long they take, and when and to what extent they are effective.
- How to predict which methods will be used in the future of the computation and thus need to be precompiled? Our results suggest that this precompilation can make a significant difference in the benefit we get out of optimizing on the server.
- Even if methods are not precompiled, can we overlap server

[4]We believe their experiments are for C or C++ programs, but the papers do not make that clear. Also, we are not sure if they enabled or disabled optimizations during compilation.

and client effectively, so as to reduce idle time? This would involve predicting which methods the client will need in the near future, based on recent demands.

- How to deal with the issue of inlining of methods? Sometimes inlining is beneficial, but it often costs significantly more than optimization without inlining, and the resulting code tends to be larger.

- How does interpretation (either instead of or in combination with compilation) compare to our configurations? Which methods should be interpreted, and which compiled, especially if we can obtain compiled versions from a server?

- How does downloading preanalyzed bytecodes compare to our configurations? To avoid the long idle times while waiting for a server to optimize, it may make sense to have the server simply annotate the bytecodes based on potentially expensive analyses. The handheld can use those annotations to optimize code if necessary.

- Does the story change significantly if the client is a laptop or notebook computer rather than a handheld?

7. CONCLUSIONS

Since high-performance Java systems spend significant time and energy compiling and optimizing code we consider moving compilation to a tethered server. We consider four main configurations: (i) the handheld compiles code itself; (ii) the handheld optimizes code itself; (iii) the server compiles the code and the handheld downloads it; (iv) the server optimizes the code and the handheld downloads it. These configurations offer different tradeoffs. Optimized (and even compiled) code is larger than bytecodes and thus will incur greater download energy costs than bytecode. Optimizations take longer than non-optimizing compilation and thus incur more energy cost. Optimized code runs faster than non-optimized code and thus uses less energy.

Our results show that there is significant benefit to moving optimizations to a server. We find that at least for longer running methods, the energy benefits of optimizations (due to faster execution time) overwhelms the costs of optimizations (greater download and optimization energy).

Acknowledgements

We thank Michael Hind and Martin Hirzel for their insightful comments on the paper.

8. REFERENCES

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1), 2000.

[2] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison Wesley, third edition, 2000.

[3] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter Sweeney. Adaptive optimization in the Jalapeño jvm. In *ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, 2000.

[4] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 83–94, Vancouver, Canada, June 2000.

[5] Michael Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio Serrano, V. C. Sreedhar, and Harini Srinivasan. The Jalapeño dynamic optimizing compiler for Java. In *ACM Java Grande Conference*, San Francisco, CA, June 1999.

[6] G. Chen, R. Shetty, N. Vijaykrishnan, M. M. Irwin, and M. Wolczko. Tuning garbage collection in an embedded environment. In *Symposium on High Performance Computer Architecture (HPCA)*, 2002. To appear.

[7] Compaq Computer Corporation. ipaq 3835 pocket pc. http://athome.compaq.com/showroom/static/iPaq/3835.asp.

[8] Compaq Computer Corporation. Wl110 product specifications. http://www.compaq.com/products/wireless/wlan/wl110.shtml.

[9] Palm Corporation. http://www.palm.com/products/palmvx/.

[10] Amer Diwan, Han Lee, Dirk Grunwald, and Keith Farkas. Energy consumption and garbage collection in low-powered computing. Submitted for publication.

[11] Keith I. Farkas, Jason Flinn, Godmar Back, Dirk Grunwald, and Jennifer Anderson. Quantifying the energy consumption of a pocket computer and a Java virtual machine. In *Proceedings of the ACM SIGMETRICS '00 International Conference on Measurement and Modeling of Computer Systems*, 2000.

[12] Jason Flinn, Dushyanth Narayanan, and M. Satyanarayanan. Self-tuned remote execution for pervasive computing. In *8th Workshop on Hot Topics in Operating Systems (HotOS-VIII*, May 2001.

[13] William R. Hamburgen, Deborah A. Wallach, Marc A. Viredaz, Lawrence S. Brakmo, Carl A. Waldspurger, Joel F. Bartlett, Timothy Mann, and Keith I. Farkas. Itsy: Stretching the bounds of mobile computing. *IEEE Computer*, 34(4):28—36, April 2001.

[14] Mahmut T. Kandemir, Narayanan Vijaykrishnan, Mary Jane Irwin, and W. Ye. Influence of compiler optimizations on system power. In *Design Automation Conference*, pages 304–307, 2000.

[15] H. S. Kim, M. J. Irwin, N. Vijaykrishnan, and M. Kandemir. Effect of compiler optimizations on memory energy. In *IEEE Workshop on Signal Processing Systems*, pages 663–672, 2000.

[16] Ulrich Kremer, Jamey Hicks, and James M. Rehg. Compiler-directed remote task execution for power management. In *Workshop on compilers and operating systems for low power (COLP'00)*, October 2000.

[17] Ulrich Kremer, Jamey Hicks, and James M. Rehg. A compilation framework for power and energy management on mobile computers. Technical Report DCS-TR-446, Rutgers University, 2001.

[18] Michael Paleczny, Christopher Vick, and Cliff Click. The Java HotSpot(TM) server compiler. In *Java Virtual Machine Research and Technology Symposium*, April 2001.

[19] Alexey Rudenko, Peter Reiher, Gerald J. Popek, and Geoffrey K. Kuenning. Saving portable computer battery power through remote process execution. *Mobile Computing and Communcations Review*, 2(1), 1998.

[20] Alexey Rudenko, Peter Reiher, Gerald J. Popek, and Geoffrey K. Kuenning. The remote processing framework for portable computer power savings. In *Proceedings of the ACM Symposium on Applied Computing (SAC99)*, San Antonio, TX, February 1999.

[21] M. Satyanarayanan. Pervasive computing: Visions and challenges. *IEEE Personal Communications*, 2001.

[22] Emin Gun Sirer, Robert Grimm, Arthur J. Gregory, and Brian N. Bershad. Design and implementation of a distributed virtual machine for networked computers. In *17th ACM Symposium on Operating System Principles (SOSP '99)*, pages 202–216, Kiawah Island, SC, December 1999.

[23] Radu Teodorescu and Raju Pandey. Using jit compilation and configurable runtime systems for efficient deployment of Java programs on ubiquitous devices. In *Ubiquitous computing 2001, LNCS 2201*, pages 76–95, 2001.

[24] V. Tiwari, S. Malik, and A. Wolfe. Compilation techniques for low energy: An overview. In *IEEE Symposium on Low Power Electronics*, 1994.

[25] V. Tiwari, S. Malik, A. Wolfe, and M. Lee. Instruction level power analysis and optimization of software. *Journal of VLSI Signal Processing*, 13(2/3):1–18, 1996.

[26] M. Valluri and L. John. Is compiling for performance==compiling for power? In *5th Annual Workshop on Interaction Between Compilers and Computer Architectures (INTERACT-5)*, January 2001.

[27] M. A. Viredaz. The Itsy Pocket Computer version 1.5: user's manual. Technical Report TN-54, Western Research Lab, Compaq Computer Corporation, July 1998.