S&N INVENT

Value Driven Performance.

LEISTUNG BLEIBT WICHTIG

# Immer schneller - aber wie?

Teil 1

Dr. Jürgen Lampe
S&N Invent GmbH

## Abstract

Seit mehr als zehn Jahren stagniert die Entwicklung der CPU-Taktfrequenz im Bereich 3-4 GHz. Durch Verbesserungen in anderen Komponenten ist es bisher gelungen, trotzdem die Gesamtleistung der Systeme weiter zu steigern. Da auch diese Potentiale nicht unerschöpflich sind, wird zukünftig die Softwareentwicklung dem Leistungsaspekt größere Beachtung schenken müssen. Dieser zweiteilige Beitrag behandelt zunächst ganz generell die verschiedenen Wege, Programme schneller laufen zu lassen. Im zweiten Teil folgen dann konkrete Vorschläge für die Java-Entwicklung in einer Welt, in der Leistungssteigerungen durch neue Hardware nicht mehr selbstverständlich sind.

## Einführung

Geschwindigkeit ist schon immer ein wichtiges Ziel der Computeranwendung gewesen. Über viele Jahre gab es als Folge der technischen Entwicklung wachsende Prozessorleistung weitgehend beiläufig. Dadurch wurde beim Programmieren eine gewisse Sorglosigkeit begünstigt, die in dem als *Reisers* bzw. *Wirthsches Gesetz* bekannten Bonmot „Die Software wird schneller langsamer, als die Hardware schneller wird.“ attackiert wird. Für die Softwareentwicklung war und ist Performance Engineering oft erst ein Thema, wenn die geforderten Werte nicht erreicht werden.

Das beginnt sich zu ändern. Seit etwa zehn Jahren steigt die Prozessor-Taktfrequenz und damit die Ausführungsgeschwindigkeit der einzelnen Operationen kaum noch an.[1] Aus Sicht der Anwender von Programmen ist diese Stagnation bisher weitgehend verborgen geblieben, weil zunächst eine ganze Reihe von technischen Möglichkeiten zur Steigerung der Systemleistung erschlossen wurden:

- Parallelverarbeitung durch Multicore-Prozessoren und Mehrprozessor-Systeme
- Prozessor-Pipelines mit spekulativer Ausführung (Preload, Prefetch)
- Größere Hauptspeicher (für z. B. In-Memory-Datenbanken)
- Größere Prozessor-Caches
- Schnellere Peripheriegeräte, z. B. SSD, USB-3

Für Java-Programme kommt hinzu, dass die JVM nicht nur von den genannten Vorteilen profitierte, sondern selbst konsequent weiterentwickelt wurde, sodass Bytecode heute selbst ohne die verbesserte Hardware spürbar schneller ausgeführt wird als vor zehn Jahren.
Inzwischen sind jedoch alle diese Quellen für die Leistungssteigerung weitgehend ausgeschöpft, und es wird immer schwieriger neue Wege zu finden.

## Leistung bleibt wichtig

Leistung war nicht nur wichtig, sie wird es bleiben und wahrscheinlich sogar noch wichtiger werden. Das ergibt sich allein schon daraus, dass die Komplexität der zu lösenden Aufgaben ständig steigt. Der Grad der Interaktivität nimmt dabei ebenso zu wie der Umfang der zu bearbeitenden Daten. Für viele Aufgabenklassen wächst der für die Lösung notwendige Berechnungsaufwand deutlich schneller als die Problemgröße. Je stärker die IT alle Bereiche durchdringt, desto anspruchsvoller werden die durch Software zu implementierenden Funktionen.

> Performance und agile Entwicklung
>
> Agiles Vorgehen vermindert - auf den ersten Blick - die Wahrscheinlichkeit, dass Performanceprobleme (zu) spät entdeckt werden. Das gilt allerdings nur, wenn
>
> - sich die entsprechenden Anforderungen im Laufe der Entwicklung nicht stark ändern und
> - keine leistungskritischen Funktionen erst in sehr späten Phasen angefordert und/oder implementiert werden.
>
> Durch die relativ *kurzsichtige* Arbeit besteht die Gefahr, dass globale Aspekte zu wenig berücksichtigt werden. Und so agil, nach beinahe Fertigstellung nochmals fast von vorn zu beginnen, ist man dann doch eher selten.

Ein anderer Grund, aus dem Leistung stärker in den Fokus gerät, ist wirtschaftlicher Natur. Computeroperationen verursachen Kosten durch Stromverbrauch und möglicherweise erforderliche Kühlung. Rechenzentren gehören mancherorts bereits zu den größten Energieverbrauchern. Ebenso wichtig ist die Energieaufnahme für die wachsende Zahl mobiler Geräte, weil dadurch die potentielle Nutzungszeit direkt beeinflusst wird. Programme, die ihre Aufgabe mit weniger Operationen erledigen, sind nicht nur schneller, sondern brauchen dafür auch weniger Energie. Unter dem Stichwort *Green IT* ist diese Erkenntnis in der Welt der Rechenzentren angekommen. In der Softwareentwicklung gibt es erste Ansätze derartige Gesichtspunkte zu berücksichtigen. [2]

Bisher werden bei der Spezifikation neuer Software Leistungskennwerte meist als *nichtfunktionale Anforderungen* geführt und oft nur recht vage formuliert. Wenn man im Notfall auf schnellere Maschinen ausweichen kann, mag das vertretbar sein. Das Problem besteht aber darin, dass solche nicht-funktionalen Anforderungen im Entwurfsprozess gern als untergeordnete Ziele behandelt werden. Angesichts der technischen Grenzen wird es jedoch zunehmend wichtiger, Leistungsziele von Anfang an in das Design einzubeziehen. Dazu ist es allerdings notwendig, sich bereits in der Requirement-Phase Gedanken über das Leistungsmodell und die wichtigsten Einflussfaktoren zu machen.

In den folgenden Abschnitten werden die verbleibenden Möglichkeiten zur Leistungssteigerung kritisch analysiert. Dabei zeigt sich, dass Leistung ein Kostenfaktor mit zunehmender Bedeutung ist, der in Konkurrenz zu anerkannten Zielen wie Robustheit, Flexibilität und Wartbarkeit tritt.

## Parallelisierung ist nicht die Lösung

Herb Sutter sieht in seinem bereits erwähnten Artikel [1] einen grundlegenden Wandel zu nebenläufigen Programmen als Konsequenz daraus, dass die Zeit rasch wachsender Prozessorleistungen („The free performance lunch“) vorbei ist. Das ist jedoch nur die halbe Wahrheit. Richtig ist, dass Parallelverarbeitung für eine ganze Reihe von Aufgaben attraktive Verbesserungen zulässt. Viele dieser Anwendungsbereiche sind allerdings bereits erschlossen worden. Es ist ja nicht das erste Mal, dass die Prozessorentwicklung vor einer vermeintlichen oder echten Grenze steht. Jedes Mal, wenn das in der Vergangenheit der Fall war, hat das Interesse an parallelen Verfahren zugenommen. Im Ergebnis ist Parallelverarbeitung heute in vielen Bereichen, beispielsweise der Bildverarbeitung und -erkennung, etabliert. Dort, wo die Ergebnisse nicht so überzeugend waren, ist man bei der weitgehend sequentiellen Verarbeitung geblieben und hat die Möglichkeiten der schnelleren Prozessoren dankbar angenommen. Denn bisher stellten sich die befürchteten Grenzen jedes Mal als scheinbare heraus, die durch neue Fertigungsverfahren überwunden werden konnten. Das ist nun offensichtlich erstmals anders. Im Moment ist keine Technologie erkennbar, die in absehbarer Zeit nochmals eine deutliche Steigerung der unmittelbaren Operationsgeschwindigkeit ermöglichen könnte. Die einzige Chance zur Leistungssteigerung scheint die Ausnutzung der verschiedenen verfügbaren Features zur parallelen Verarbeitung zu bieten.

Wie gerade erklärt, ist die Parallelisierung von Algorithmen kein unbestelltes Feld. Vielmehr hat schon eine starke Auslese stattgefunden. Vieles, was leicht parallelisierbar ist, wird bereits parallel verarbeitet, oder entsprechende Lösungen sind bekannt. Für den größten Teil dessen was übrig ist, erweist sich die Parallelisierung als prinzipiell schwieriges Problem. In Teilbereichen wird es weitere Fortschritte geben, aber eine generelle Lösung ist nicht zu erwarten. Das grundlegende Dilemma sei an einem anschaulichen Beispiel illustriert: dem Bau einer Eisenbahnstrecke. Theoretisch kann ein solcher Bau durch Parallelarbeit fast beliebig beschleunigt werden, indem zum Beispiel alle paar Meter ein Bautrupp stationiert wird, der nur diesen kurzen Abschnitt bearbeiten muss. Man sieht sofort das wesentliche Problem paralleler Arbeit – das Bereitstellen der Ressourcen. Die Trupps müssen mit ihrer Ausrüstung und dem Material verteilt und wieder eingesammelt werden. Das Verhältnis von Aufwand zu Nutzen ist nur bei einer sehr gemäßigten Parallelisierung vertretbar. Eine zweite Schlussfolgerung betrifft die Art der Arbeit. In der Frühzeit des Eisenbahnbaus, als Schaufel und Schubkarre die wichtigsten Werkzeuge waren und die Gehzeit der Arbeiter nicht bezahlt wurde, hat dieser Ansatz für die Planierung der Strecke sogar funktioniert. Das Produkt (die Bahnstrecke) ist mit der Zeit aber komplizierter geworden. Signalkabel, Oberleitungen und stoßlos verschweißte Schienen erfordern eine höhere Kooperation zwischen den Trupps und manchmal eine Sequentialisierung (Kabel). Es ist eine über das Beispiel hinausgehende Beobachtung, dass

anspruchsvollere Aufgaben tendenziell die Parallelisierung erschweren. Koordination und Kommunikation zwischen parallelen Prozessen sind nicht kostenlos. Der Einfluss sequentieller Arbeitsabschnitte auf die erreichbare Beschleunigung ist erheblich.

> Amdahls Gesetz
>
> Mit diesem Namen wird ein 1967 von Gene Amdahl publizierter Zusammenhang für die Abschätzung der möglichen Beschleunigung von Programmen durch Parallelisierung bezeichnet. Praktisch relevant ist vor allem der daraus ableitbare, eigentlich triviale Sachverhalt, dass ein parallelisiertes Programm mindestens so viel Zeit verbraucht, wie der verbleibende nichtparallelisierte (sequentielle) Teil ($s<1$). Daraus ergibt sich für die erreichbare Beschleunigung $B$ als Grenzwert:
>
> $$B = 1 / s.$$
>
> Das heißt, wenn 20% der Laufzeit auf nichtparallelisierbaren Code entfallen, kann bestenfalls eine Beschleunigung um den Faktor fünf $(=1/0,2)$ erreicht werden. Der tatsächliche Wert liegt immer niedriger, da die Zeiten für den parallelisierten Teil sowie für eventuelle Synchronisierungen bei diesem Grenzwert unberücksichtigt bleiben.

Generell wird das Schreiben paralleler Programme dadurch erschwert, dass es keine einheitliche Hardware gibt und der Code dadurch Gefahr läuft, maschinenabhängig zu sein. Ein zusätzliches Problem für die kleinteilige Parallelisierung mit Mehrkern-Mehrprozessor-Architekturen erwächst aus den beteiligten Caches, die je nach Schicht (L1, L2, L3) gemeinsam oder exklusiv genutzt werden. In ungünstigen Konstellationen können notwendige Aktualisierungen einzelner Cache-Lines wegen unbeabsichtigter Synchronisierungen zu erheblichen Verzögerungen führen. Auf Quellcode-Ebene kann auf solche Zuordnungen praktisch kein Einfluss genommen werden, sodass die Ausführungszeiten scheinbar zufällig variieren.

Das war die gern übersehene Hälfte der Wahrheit. die andere lautet: Parallelisierung ist die geeignete Technologie für alle Probleme, die sich relativ leicht in unabhängige, aber nicht zu kleine Teilaufgaben mit geringen Anforderungen an Kommunikation und Synchronisation zerlegen lassen. Je weniger diese Voraussetzungen erfüllt sind, desto schwieriger wird die Parallelisierung und desto geringer wird der zu erwartende Effekt. Als Allheilmittel für Performanceprobleme scheidet die Parallelisierung daher aus.

## Automatische Optimierungen

Um etwaigen Missverständnissen vorzubeugen, wird hier der Begriff Optimierung klar vom Performance-Tuning unterschieden. Optimierung ist das ingenieurmäßige Bemühen, eine Aufgabe unter definierten Bedingungen so zu lösen, dass alle Anforderungen, insbesondere auch bezüglich Kosten und Zeit, bestmöglich erfüllt werden. Das Ergebnis ist typischerweise ein Kompromiss. Im Gegensatz dazu ist Tuning oder *Feinanpassung* der Versuch, die Leistung in einem Teilbereich zu

verbessern, auch auf die Gefahr von Verschlechterungen in anderen Bereichen hin – solange die unabdingbare Funktionalität erhalten bleibt.

Für Software war geringer Bedarf an den beiden wichtigsten Ressourcen Prozessorzeit und Speicher von Anfang an wichtig. Daneben spielten und spielen die Entwicklungskosten eine Große Rolle. Weil Optimierung aufwendig ist, gab es schon früh Bemühungen um eine zumindest teilweise Automatisierung. Leider zeigt es sich, dass die besonders wirksamen „aggressiven“ Verfahren bisweilen die Semantik verändern oder dass die notwendigen Vorbedingungen nicht vollständig verifizierbar sind. Beides schließt eine standardmäßige Verwendung aus. Bei Java ist die Sache noch schwieriger. Java-Programme sind prinzipiell offen. Über `Class.forName(...)` kann jederzeit zusätzlicher Code ergänzt werden. Dieser nachgeladene Code muss dazu beim Programmstart noch gar nicht existieren oder zugreifbar sein. Somit ist eine vollständige statische Analyse des Quellcodes als Voraussetzung jeder Ex-ante-Optimierung prinzipiell ausgeschlossen. Ein weiterer Nachteil der statischen Verfahren ist es, dass es fast unmöglich ist, für die Leistung wesentliche Codeabschnitte von solchen zu unterscheiden, die selten oder gar nicht ausgeführt werden.

Mit der dynamischen Optimierung durch die Hotspot-VM ist es gelungen, gleich beide Probleme elegant zu lösen Die Beschleunigungsbemühungen werden dabei auf häufig durchlaufene Abschnitte, die *Hotspots*, beschränkt. Erfahrungsgemäß ist das nur ein relativ kleiner Anteil. Die auf diesem Weg erreichten Ergebnisse sind beeindruckend, aber auch sie haben ihren Preis.

Zunächst verursacht die dynamische Optimierung zusätzlichen Laufzeitaufwand. Es müssen Daten über die laufende Anwendung gesammelt und die resultierenden Optimierungen ausgeführt werden. Unter Umständen ist es auch notwendig, bestimmte Schritte zurückzunehmen (zu *deoptimieren*), weil durch neu zu ladende Klassen, die ursprünglichen Voraussetzungen nicht mehr erfüllt sind. Auch wenn diese Aktivitäten in eigenen Threads mehr oder weniger parallel erfolgen, erfordern sie Prozessorzyklen, die dem eigentlichen Programm nicht mehr zur Verfügung stehen.

Da die Hotspot-Optimierung zunächst statistische Daten sammeln muss, ist sie vorrangig für länger laufende Prozesse sinnvoll.

Eine andere Auswirkung der dynamischen Optimierung ist ihr quasi nichtdeterministischer Charakter. Unterschiedliche Daten führen zu gewöhnlich unterschiedlichen Befehlssequenzen, was zur Folge hat, dass die Schwellwerte in variierender Folge erreicht werden. Außerdem muss man damit rechnen, dass möglicherweise andere Code-Teile benötigt werden. Wenn man großes Pech hat, kann sich die Performance einer Anwendung allein dadurch spürbar verschlechtern (oder auch verbessern), dass sich die Datenstruktur ändert. Hinzu kommt, dass die Einflüsse der Hardware auf die zeitlichen Abläufe der Threads innerhalb der VM dazu führen, dass selbst bei identischen Daten unterschiedlich optimiert wird.

Schließlich sei noch auf ein prinzipielles Problem der automatischen Optimierung hingewiesen. Sie bezieht sich fast nur auf die Ausführung von Operationen. Bei Sprachen wie Java entsteht jedoch ein erheblicher Teil des Aufwands durch die Speicherverwaltung, d. h. das Erzeugen von Objekten und die Speicherbereinigung. Diesbezügliche Verbesserungen sind sehr viel schwieriger zu realisieren. Bekannt ist hier vor allem die auf einer Escape-Analyse beruhende Verlagerung kurzlebiger Objekte in den Stack. Dadurch entfällt die Behandlung durch den Garbage-Collector. Wie es scheint, ist der Effekt allerdings wenig auffällig.

Escape-Analyse

Darunter versteht man ein bereits 1999 publiziertes Verfahren [5], mit dessen Hilfe bei der Allokation ermittelt wird, ob ein Objekt möglicherweise außerhalb des erzeugenden Befehlsstroms (Thread) sichtbar sein kann. Wenn beispielsweise festgestellt wird, dass ein Objekt nur von lokalen Variablen referenziert wird, kann es problemlos wie diese im lokalen Stack angelegt werden. Denn mit dem Verschwinden dieser Variablen beim Rücksprung (Return) ist das Objekt nicht mehr zugreifbar und damit obsolet. Für Objekte, die den jeweiligen Thread nicht verlassen (escape), können außerdem die Synchronisierungsoperationen entfernt werden.
Seit Java SE 6.23 ist diese Funktion standardmäßig aktiv.

Bei aller Anerkennung der Leistung der Hotspot-VM - und die ist ohne Zweifel bemerkenswert - muss man feststellen, dass die Zeit der wirklich großen Fortschritte vorbei ist. Mittlerweile steht durchaus die Frage im Raum, ob weitere Verbesserungen noch sinnvoll oder überhaupt noch möglich sind, weil Grenzen der Hardware erreicht werden.

## Effizienter Code

Zusammengefasst lässt das bisher Gesagte nur den Schluss zu, dass die Quellen impliziter Leistungssteigerungen, d. h. solcher, um die sich Entwickler nicht speziell kümmern müssen, weitgehende ausgeschöpft sind. Dem gegenüber stehen die eingangs beschriebenen wachsenden Anforderungen. In dieser Situation kann die Lösung nur darin bestehen, dem Ziel guter Leistung im Softwareentwicklungsprozess mehr Beachtung zu schenken und bewusst darauf hinzuarbeiten. Es ist ein ganz natürlicher Vorgang, dass mit zunehmendem Reifegrad einer Produktion zunächst rein nicht-funktionale Ziele in die Rolle von funktionalen wachsen. Beispiel Auto: Da steht schon lange nicht mehr nur auf der Agenda, dass es fahren muss. Da kamen schon bald Geschwindigkeit und Beschleunigung dazu und inzwischen wird wohl keine Entwicklung mehr gestartet, ohne Verbrauchsvorgaben. Ganz so weit ist die Softwareentwicklung auf die Leistung bezogen im Allgemeinen noch nicht.

Um der Leistung einen höheren Stellenwert zuzuweisen, sind Veränderungen von zwei Seiten erforderlich. Einerseits muss bei den Entwicklern das Wissen darüber verbessert werden, welche Kosten die einzelnen Programmkonstrukte zur Laufzeit verursachen. Gleichzeitig müssen die Tugenden guter Ingenieurarbeit vermittelt werden, die vor allem darin bestehen, immer das

Gesamtziel im Auge zu behalten und für jede Teilaufgabe den insgesamt optimalen Kompromiss zwischen Funktion und Kosten zu finden.

Um diesem Ziel näher zu kommen wird eine breitere Sicht auf das Performance Engineering gebraucht. Derzeit kommt diese Disziplin meist erst dann in den Fokus, wenn Leistungsziele nicht erreicht werden: *The conventional approach to performance is to ignore it until deployment time. However, many, if not most, performance problems are introduced by specific architecture, design, and technology choices that you make very early in the development cycle. After the choices are made and the application is built, these problems are very difficult and expensive to fix.* [3] Das heißt gewissermaßen *Performance by design*. Beim Versuch, diesen Ansatz in die Praxis umzusetzen, zeigt sich allerdings ein Dilemma. Richtigerweise wird stets darauf hingewiesen, Leistungsuntersuchungen nur auf der Basis von Messungen vorzunehmen. Zum Messen wird allerdings lauffähige Software benötigt, die in den frühen Phasen noch nicht vorhanden ist. Als Ausweg bietet es sich an, durch Erfassung und Auswertung großer Mengen von empirischen Daten Erkenntnisse zu gewinnen, die akzeptable Prognosen in der Entwurfsphase erlauben. Leider ist das Performance Engineering noch nicht in der Lage, einen entsprechend aufbereiteten Fundus von Erfahrungen bereitstellen zu können. Dafür sind ganz objektive Schwierigkeiten verantwortlich, weil das Sammeln der Daten aufwendig ist und die Forscher nicht wie Botaniker einfach über eine Wiese gehen können, um ihre Untersuchungsobjekte einzusammeln. Intern werden Analysen nur soweit ausgeführt, wie zur Behebung eines Problems notwendig ist. Nachfolgend veröffentlichte Verallgemeinerungen sind leider äußerst selten. Man muss auch sehen, dass derartige Aktivitäten trotz ihrer Wichtigkeit derzeit weder in der Wirtschaft noch in der Wissenschaft sonderlich hoch angesehen sind. Aber solche Daten werden dringend gebraucht. Es ist sicher, dass die Prinzipien leistungsgerechten Programmierens zukünftig gleichberechtigt neben denen für wartungsfreundlichen Code stehen werden.

Es gibt jedoch für Entwickler keinen Grund die Hände in den Schoß zu legen bis entsprechende Erfahrungen publiziert werden. Denn natürlich gibt es Zusammenhänge, die so elementar sind, dass sie unabhängig von den konkreten Bedingungen immer gelten. So wie es beim Auto einen naturgesetzmäßigen Zusammenhang zwischen Fahrzeugmasse und Verbrauch gibt, gilt für Programme beispielsweise, dass weniger auszuführende Operationen in der Regel zu höherer Leistung führen. Allerdings sind die Verhältnisse bei Software deutlich komplizierter, sodass eine genauere Betrachtung gefragt ist. Darüber hinaus sind Leistungsaussagen zunehmend von Eigenschaften der Hardware abhängig. Diese Themen werden im zweiten Teil ausführlicher diskutiert werden.

Ebenso wichtig wie die Integration der Leistungsproblematik in den Entwicklungsprozess ist es, bei den Auftraggebern einer Software das Bewusstsein zu schaffen, dass entsprechende Parameter ein

integraler Bestandteil der Aufgabenbeschreibung sein müssen. Das Ende der quasi automatischen Leistungssteigerungen bedeutet letztlich auch das Ende des verbreiteten *One-fits-all*-Ansatzes, nicht nur in Projekten, sondern auch bei Bibliotheken und Frameworks. Wenn man nicht einfach einen schnelleren Prozessor einsetzen kann, weil es den nicht gibt, muss man sich rechtzeitig Gedanken um das benötigte Leistungsprofil machen, und das auch in der Kostenkalkulation berücksichtigen. Die Erfahrung zeigt, dass der Ermittlung von Nutzungsdaten deutlich weniger Aufmerksamkeit geschenkt wird als der fachlichen Analyse.

Wenn die erwartete Leistung beim Entwickeln berücksichtigt werden soll, muss sie erst einmal bekannt sein. Software entwickeln heißt nicht zuletzt, ständig Auswahl-Entscheidungen zu treffen, wie einzelne Aufgaben zu lösen sind. Je genauer die Erwartungen spezifiziert sind, umso besser können sie berücksichtigt werden. Deshalb sollten zum Beispiel Use-Cases nicht nur funktional, sondern auch in ihrer Häufigkeit definiert sein, entweder absolut oder mit Bezug auf übergeordnete Fälle. Nur so kann man erreichen, dass die Entwicklung von Anfang an auf die für die Leistung wichtigen Teile konzentriert wird. Letztlich ist es auch eine Kostenfrage, den Aufwand auf die richtigen Stellen zu lenken und unnötige Bemühungen zu vermeiden.

Für die Auftraggeber ist das eine neue und ungewohnte Herausforderung. Sie werden sich daran gewöhnen müssen, dass eine Software gezielt für ein vorgegebenes Profil entwickelt wird. Wenn sich herausstellt, dass die tatsächlichen Anforderungen nicht (mehr) diesem Profil entsprechen, muss – ganz analog zu geänderten (kern-)funktionalen Anforderungen – gegebenenfalls die Software oder ein Teil davon neu entwickelt werden. Anders werden sich hohe Leistungsanforderungen in absehbarer Zukunft nicht mehr realisieren lassen.

## Schlussfolgerungen

Das Ziel dieses Beitrages besteht darin, zu begründen, warum effizienter Code schnell an Bedeutung gewinnen wird. Dazu sind Ausweitung und Aufwertung des Performance Engineering sowie die stärkere Einbindung in den Softwareentwicklungsprozess unumgänglich. Das heißt für alle Beteiligten vom Requirement Engineering bis zum Programmieren, dass jeweils stufenspezifische Kenntnisse zur leistungsgerechten und anforderungsadäquaten Entwicklungsarbeit vorhanden sein und angewendet werden müssen. Die Notwendigkeit, Leistungsziele gegen konkurrierende Erfordernisse wie kurze Entwicklungszeit, Flexibilität oder Wartbarkeit abwägen zu müssen, erhöht die Herausforderung.

Natürlich wird diese Aufgabe nicht sofort vor jedem stehen. Es wird auch weiterhin (kleine?) Fortschritte bei der impliziten Leistungssteigerung geben. Vielleicht gelingt es in naher Zukunft doch noch einmal, eine spürbare Verbesserung in der skalaren Prozessorleistung zu realisieren [4]. Dass würde die Grenze verschieben, hinter der die beschriebenen Überlegungen unverzichtbar sind.

Unabhängig davon wird es viele Aufgaben geben, die keine extreme Leistung erfordern oder sich passend zerlegen und dezentralisieren lassen. Am besten (wenn auch nicht für uns Entwickler) ist es, wenn gründliches Überlegen dazu führt, eine Aufgabe als überflüssig zu erkennen. Trotzdem sollte man sich mit dem an der Leistung orientierten Entwickeln befassen, um für die Zukunft gewappnet zu sein. Mehr als bei manchen anderen Techniken kommt es hierbei auf das Üben und Probieren an, und nicht zu vergessen, das Messen. Es sollte keine Möglichkeit ausgelassen werden, sich über Leistungsdaten der entwickelten Software in der Praxis zu informieren. Nur so gewinnt man die nötige Erfahrung.

## Über S&N Invent GmbH

Die S&N Invent GmbH bündelt die Leistungen und das Angebot der beiden Gruppengesellschaften S&N AG (Paderborn) und A:gon Solutions GmbH (Eschborn). Gemeinsam sind sie zukünftig unter dem Dach der S&N Invent GmbH noch leistungsfähiger und damit für Kunden und Partner attraktiver – dank einem gemeinsamen und damit wesentlich erweiterten Portfolio, verbesserter überregionaler Präsenz, einer deutlichen personellen Verstärkung und sich hervorragend ergänzender Kompetenzen.

Sowohl die A:gon Solutions GmbH als auch die S&N AG sind langjährig erfolgreiche und etablierte IT-Unternehmen. Insgesamt beschäftigt die S&N Invent ca. 350 feste und freie Mitarbeiterinnen und Mitarbeiter an sieben Standorten in Deutschland. So ist sie hervorragend aufgestellt, um die anstehenden Herausforderungen bei der IT-gestützten Optimierung von Geschäftsprozessen, in der Implementierung neuer und der Renovierung bestehender Softwarelösungen sowie in allen Bereichen der IT-Infrastruktur zusammen mit ihren Kunden und Partnern erfolgreich bewältigen zu können.

## Autor

Jürgen Lampe ist IT-Berater bei der A:gon Solutions GmbH, einem Unternehmen im Verbund der S&N Invent GmbH in Frankfurt. Seit mehr als 15 Jahren befasst er sich mit Design und Implementierung von Java-Anwendungen im Bankenumfeld. Sein Interesse gilt besonders Fachsprachen (DSL) und Werkzeugen für deren Implementierung.

## Links & Literatur

[1] H. Sutter, The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software *Dr. Dobb's Journal,* **2005***, 30*, http://www.gotw.ca/publications/concurrency-ddj.htm

[2] Chr. Bunse, S. Stiemer: On the Energy Consumption of Design Patterns. 2. Workshop Energy Aware Software-Engineering and Development (EASED@BUIS) 25.04.2013, Oldenburg, http://www.se.uni-oldenburg.de/documents/CBSS_eased2.pdf

[3] MDN Fundamentals of Engineering for Performance, https://msdn.microsoft.com/en-us/library/ff647781.aspx

[4] The POET (Planar Opto Electronic Technology) platform, http://www.poet-technologies.com/

[5] Jong-Deok Choi et al., Escape Analysis for Java, Proc. of ACM SIGPLAN OOPSLA Conf. 1999